Design-time performance analysis of component-based real-time systems

Design-Time Performance Analysis of Component-Based Real-Time Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen op
dinsdag 22 december 2009 om 16.00 uur

door

Yahor Bondarau

geboren te Zhodino, Wit-Rusland (Belarus)

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. P.H.N. de With
en
prof.dr. J.J. Lukkien

Copromotor:
dr. M.R.V. Chaudron

A catalogue record is available from the Eindhoven University of Technology Library

Bondarau, Yahor.

Design-Time Performance Analysis of Component-Based Real-Time Systems
by Yahor Bondarau. - Eindhoven : Technische Universiteit Eindhoven, 2009.
Proefschrift. - ISBN 978-90-386-2126-5
NUR 918

Keywords: performance prediction / real-time systems /
component-based software engineering.
Subject headings: real-time systems / software design /
software quality.

Design-Time Performance Analysis of Component-Based Real-Time Systems

Yahor Bondarau

Committee members:

prof.dr.ir. P.H.N. de With (TU Eindhoven, promoter)
prof.dr. J.J. Lukkien (TU Eindhoven, promoter)
dr. M.R.V. Chaudron (LIACS Leiden, copromoter)
prof.dr.ir. A.P.C.M. Backx (TU Eindhoven, chairman)
prof.dr. M.G.J. van den Brand (TU Eindhoven)
prof.dr. V. Cortellessa (Universita dell'Aquila Italy)
prof.dr. I. Crnkovic (Malardalen University Sweden)
ir. J.H.A. Gelissen (Philips Research Labs Eindhoven, advisor)
prof.dr. P. Pettersson (Malardalen University Sweden)

The research work reported in this thesis was supported by two international projects in the framework of the ITEA programme: Space4U and Trust4All.

Acknowledgements

You are holding a book that results from four years of research work. However, as a wise proverb says: *The process, not the final result, is what brings the biggest pleasure for a creative person*, the most difficult and interesting was the path to this result. Whether there were seconds of despair and enlightening, periods of self-reflection and hard implementation of ideas, as well as the junction points, where a smart decision should be taken to proceed successfully, in all these moments I felt the support and understanding from the people that I worked and lived with during this path. I am very grateful to all of them.

First and foremost, I would like to thank my promotor and the leading star in my research career prof.dr. Peter H.N. de With, who accepted me as a young PhD student and opened up the gates to the scientific world. After six years of working together, I am still impressed by his creativity, performance and enthusiasm, that positively influenced me at the beginning of my PhD. His technical expertise, world-true examples and our endless discussions helped me to understand how things can be done in the most efficient way. Besides this, his openness and a great sense of humor make me feeling at home, while working in the VCA Lab at the TU/e.

I also would like to express my sincere gratitude to my second promotor prof.dr. Johan J. Lukkien and copromotor dr. Michel R.V. Chaudron. Their knowledge, experience and willingness to help allowed me to obtain a deep insight into the world of system architectures, real-time systems and CBSE technologies. My architectural and analysis skills were formed by an objective feedback loop from their side. Johan and Michel, thank you for that!

I would like to thank all the members of the Doctorate Committee, Professors Vittorio Cortellessa, Ivica Crnkovic, Mark van den Brand and Paul Pettersson for their valuable and detailed feedback on this thesis. This feedback helped a lot to improve the thesis quality.

Special words of gratitude go to Jean Gelissen and Hugh Maaskant (Philips Research), who heavily supported me in my 'strange' performance prediction method for the ROBOCOP architecture during the Space4U and Trust4All international projects. Their invitation to co-author them in the two CBSE tutorials was a great credit, which set my motivation one level higher.

This work would never be completed without Harold Weffers, Director of the Software Technology (OOTI) programme, who invited me for the postmaster study and provided the access to the European IT world. His vision on the software engineering discipline and life in The Netherlands in general, that he expressed in a metaphoric and intriguing way, left a great impact on my working and living principles.

I am grateful to my colleagues and friends in the SAN and VCA groups at the TU/e for giving a permanent and warm inspiration during this PhD period. Especially, I am thankful to Goran (The Bastardo) Petrovic and Milan

Pastrnak, with whom we performed a lot of beer-to-beer reviews of the achieved results and ideas.

I also want to express my gratitude to my friends Pashka, Luda, Oleksii, Andrew, as well as to Kostya and Serge for making this tough concept of life more cosy, relaxed and joyful.

Мои дорогие Мама и Папа, несмотря на обилие иностранных букв, эта книга - для вас. Вы можете гордиться мной так же как я всегда гордился вами. Благодаря вашей заботе, поддержке и любви, а также привитому 'теоретическому' трудолюбию, я смог успешно пройти все ступеньки на пути к этой книге. Хочу также поблагодарить сестру Иринку, Надежду Ивановну и Виктора Павловича за поддержку и понимание моих ночных 'бдений' у компьютера.

Most importantly, I would like to thank my dear wife Yevgenia for her endless support and understanding. Without her love, I would not be able to fight with the difficulties experienced during this period. Being surrounded by the wisdom and patience, that is what one can only dream about.

Samenvatting

Prestatie- of performancemetrieken van actuele real-time systemen behoren tot de meest uitdagende eigenschappen om te specificeren, voorspellen en meten. Performanceëigenschappen zijn afhankelijk van diverse factoren, zoals omgevingscontext, belastingsprofiel, middleware, besturingssysteem, hardware platform en het verdelen van de interne rekenkracht. Het niet voldoen aan performanceëisen veroorzaakt vertraging, kostenoverschrijdingen en zelfs het vroegtijdig afbreken van projecten. Om deze performance gerelateerde projectproblemen te vermijden moeten de performanceëigenschappen reeds worden bepaald en geanalyseerd in de vroege ontwerpfases van een project.

In dit proefschrift worden de principes gebruikt van het componentengebaseerd ontwerpen van software (CBSE), waardoor het mogelijk is om softwaresystemen te construeren met behulp van de individuele componenten. Het voordeel van CBSE is dat de individuele componenten afzonderlijk kunnen worden gemodelleerd, (her-)gebruikt en verhandeld. Het hoofddoel van dit proefschrift is een methode te ontwikkelen die het mogelijk maakt om de performanceëigenschappen van een systeem te voorspellen, gebaseerd op de performanceëigenschappen van de individuele componenten. De voorspellingsmethode is geschikt voor snelle prototyping en performanceanalyse van de systeemarchitectuur of gerelateerde alternatieven, zonder het daadwerkelijk implementeren en testen van deze alternatieven. De onderliggende onderzoeksvragen hiervoor zijn als volgt. Hoe moeten de gedrags- en performanceëigenschappen van individuele componenten worden gespecificeerd, teneinde een automatische compositie van deze eigenschappen mogelijk te maken voor een analyseerbaar model van het complete systeem? Hoe moeten de modellen van de individuele componenten worden gesynthetiseerd voor een model van het complete systeem op een automatische wijze, zodanig dat het resulterende systeemmodel kan worden geanalyseerd met betrekking tot de performanceëigenschappen?

Het proefschrift presenteert een nieuw raamwerk genaamd DeepCompass, dat het concept realiseert van een voorspelbaar samenstel van componenten voor alle fasen van het systeemontwerp. De componentmodellen van de afzonderlijke softwarecomponenten en hardwareblokken vormen de hoekstenen van het raamwerk. De modellen zijn gespecificeerd tijdens het ontwikkelen van de componenten en komen beschikbaar als een zogenaamd componentenpakket. In de compositiefase van de componenten worden de modellen van de constituerende componenten gesynthetiseerd in een executeerbaar systeemmodel. Omdat de inhoud van het proefschrift zich concentreert op performanceëigenschappen, introduceert de auteur diverse performancegerelateerde typeringen voor componentmodellen, zoals gedrags-, performance- en hulpbronmodellen. De dynamica van de systeemexecutie wordt beschreven in scenariomodellen. Het kernvoordeel van deze modellen is dat, met behulp van het gedrag van de individuele componenten en de scenariomodellen, het gedrag van het com-

plete systeem wordt gesynthetiseerd in het eerdergenoemde executeerbare systeemmodel. De daaropvolgende simulatiegebaseerde analyse van het verkregen executeerbare systeemmodel levert waarden voor de applicatiespecifieke en systeemspecifieke performanceëigenschappen.

Ter ondersteuning van de performanceanalyse heeft de auteur een set van CARAT software hulpprogramma's ontwikkeld die voorziet in algoritmen voor automatische modelsynthese en simulatie. Daarnaast bevat de CARAT set ook grafische hulpprogramma's voor het ontwerpen van alternatieve architecturen en het visualiseren van de verkregen performanceëigenschappen.

Het onderzoek beschrijft ook een empirische casestudie naar het gebruik van scenario's in de industrie voor het analyseren van systeemperformance in een vroegtijdige ontwerpfase. Deze studie heeft aangetoond dat architecten in de industrie intensief gebruik maken van scenario's voor performanceëvaluatie. De auteur heeft, gebruik makend van de gegevens van de architecten, een verzameling van richtlijnen opgesteld voor de identificatie en het gebruik van performance-kritische scenario's.

In het laatste deel van dit proefschrift wordt het DeepCompass raamwerk gevalideerd met het uitvoeren van drie casestudies voor de voorspelling van de performance van real-time systemen: een MPEG-4 video decoder, een navigatiesysteem voor auto's en een JPEG toepassing. Voor elke casestudie zijn modellen voor de individuele componenten geconstrueerd, de software/hardware architectuur is gedefinieerd, en de CARAT hulpprogramma's zijn gebruikt voor het synthetiseren en simuleren van het executeerbare systeemmodel. De simulatie heeft de voorspelde performanceëigenschappen opgeleverd, die later zijn vergeleken met de werkelijke performanceëigenschappen van het daadwerkelijk gerealiseerde systeem. Met betrekking tot het gebruik van de hulpbronnen en de gemiddelde tijdsvertraging per taak is aangetoond dat de fout in de voorspelling varieert binnen 30% van de werkelijke performance. Wat betreft de piekbelasting per processorknooppunt werd gemeten dat de actuele waarden soms drie keer groter waren dan de voorspelde waarden.

Op grond van het voorgaande kan worden geconcludeerd dat het raamwerk effectief is in het snel realiseren van een prototype voor een architectuur en de performanceanalyse van een compleet systeem. In de casestudies is gemiddeld niet meer dan 4-5 dagen besteed aan een complete ontwikkelcyclus, inclusief het ontwerpen van verschillende alternatieve architecturen. Het raamwerk is breed toepasbaar omdat het om kan gaan met verschillende architectuurstijlen. Een conceptuele beperking van het raamwerk is dat het impliciet aanneemt dat de modellen van individuele componenten al beschikbaar zijn bij de ontwerpfase.

Summary

In current real-time systems, performance metrics are one of the most challenging properties to specify, predict and measure. Performance properties depend on various factors, like environmental context, load profile, middleware, operating system, hardware platform and sharing of internal resources. Performance failures and not satisfying related requirements cause delays, cost overruns, and even abandonment of projects. In order to avoid these performance-related project failures, the performance properties should be obtained and analyzed already at the early design phase of a project.

In this thesis we employ principles of component-based software engineering (CBSE), which enable building software systems from individual components. The advantage of CBSE is that individual components can be modeled, reused and traded. The main objective of this thesis is to develop a method that enables to predict the performance properties of a system, based on the performance properties of the involved individual components. The prediction method serves rapid prototyping and performance analysis of the architecture or related alternatives, without performing the usual testing and implementation stages. The involved research questions are as follows. How should the behaviour and performance properties of individual components be specified in order to enable automated composition of these properties into an analyzable model of a complete system? How to synthesize the models of individual components into a model of a complete system in an automated way, such that the resulting system model can be analyzed against the performance properties?

The thesis presents a new framework called DeepCompass, which realizes the concept of predictable assembly throughout all phases of the system design. The cornerstones of the framework are the composable models of individual software components and hardware blocks. The models are specified at the component development time and shipped in a component package. At the component composition phase, the models of the constituent components are synthesized into an executable system model. Since the thesis focuses on performance properties, we introduce performance-related types of component models, such as behaviour, performance and resource models. The dynamics of the system execution are captured in scenario models. The essential advantage of the introduced models is that, through the behaviour of individual components and scenario models, the behaviour of the complete system is synthesized in the executable system model. Further simulation-based analysis of the obtained executable system model provides application-specific and system-specific performance property values.

To support the performance analysis, we have developed a CARAT software toolkit that provides and automates the algorithms for model synthesis and simulation. Besides this, the toolkit provides graphical tools for designing alternative architectures and visualization of obtained performance properties.

We have conducted an empirical case study on the use of scenarios in the industry to analyze the system performance at the early design phase. It was found that industrial architects make extensive use of scenarios for performance evaluation. Based on the inputs of the architects, we have provided a set of guidelines for identification and use of performance-critical scenarios.

At the end of this thesis, we have validated the DeepCompass framework by performing three case studies on performance prediction of real-time systems: an MPEG-4 video decoder, a Car Radio Navigation system and a JPEG application. For each case study, we have constructed models of the individual components, defined the SW/HW architecture, and used the CARAT toolkit to synthesize and simulate the executable system model. The simulation provided the predicted performance properties, which we later compared with the actual performance properties of the realized systems. With respect to resource usage properties and average task latencies, the variation of the prediction error showed to be within 30% of the actual performance. Concerning the pick loads on the processor nodes, the actual values were sometimes three times larger than the predicted values.

As a conclusion, the framework has proven to be effective in rapid architecture prototyping and performance analysis of a complete system. This is valid, as in the case studies we have spent not more than 4-5 days on the average for the complete iteration cycle, including the design of several architecture alternatives. The framework can handle different architectural styles, which makes it widely applicable. A conceptual limitation of the framework is that it assumes that the models of individual components are already available at the design phase.

Contents

Chapter 1

Introduction

The more unpredictable the world is the more we rely on predictions.
Steve Rivkin

1.1 Preliminaries and Background

Real-time systems are spreading to increasingly more fields and their scope and complexity have grown dramatically in the past two decades. Real-time systems are heavily used in application domains such as avionics, automotive, medicare, multimedia, consumer electronics and surveillance. The main reason for this popularity is that these systems are able to carry out its functionality under guaranteed timing constraints. Furthermore, they are often built to ensure robustness and safety requirements.

Regarding the timing constraints, real-time systems are classified into hard or soft real-time systems. In *hard* real-time systems, e.g. an anti-lock breaking system, a missed deadline leads to disastrous consequences such as loss of life or property. A *soft* real-time system, such as a multimedia streaming application, tolerates deadline misses because they lead only to deterioration of a provided service quality.

Real-time systems are generally executing within a nondeterministic and highly concurrent environment, such as the above-mentioned anti-lock breaking system within a car. This environment generates streams of asynchronous and concurrent events having different characteristics, i.e., periodic, sporadic and aperiodic. The system should react to these events in a predefined timely manner. Moreover, real-time systems are built to execute concurrently in order to maximize their responsiveness, as well as to use their computing resources efficiently. This requires careful decisions made on the principles of the computing resource sharing with consequent mechanisms for monitoring and enforcement

of these principles. Last but not least, performance itself is a pervasive quality of the system. Every system aspect can affect the performance, from the software itself to all underlying layers, such as operating system, middleware, hardware, communication networks. The performance properties depend on multiple aspects, such as availability of hardware resources, input load, task blocking and interleaving. As a result, the latency of a real-time task may vary over time: the same execution trace of a system can be faster or slower, depending on the availability of a processor cache, or the congestion rate of a communication line.

These characteristics make the design of real-time systems complicated and they impose challenging problems to be solved during system development. These problems often result in project schedule delays, cost overruns, failures on deployment, and even abandoning of projects. To avoid or mitigate these risks, performance requirements should be carefully considered and addressed already at the early architectural phases of a project.

At these early architectural phases, the software implementations and hardware platforms for a system are often not yet available for an architect. In such cases, assessing performance properties of a future system becomes an even harder task. The way-out solution is to apply the principles of model-driven architecture (MDA) [31] for design and analysis of a real-time system. Apart from the source-code generation possibility, MDA enables specification of each software and hardware module of a system as a set of models. These models serve as an input to an analysis engine that processes this input using certain analysis algorithms, and provide an architect with predictions on system quality attributes, including performance.

Apart from the problem of assessing and handling the strict performance requirements for real-time systems, another challenge arises from the business deployment side. Namely, companies should satisfy requirements for low production costs, short time-to-market and high maintainability of their products. The Component-Based Software Engineering (CBSE) discipline aims at addressing these requirements. It enables development of software components and building systems out of pre-existing individual components. The underlying paradigm is that individual components are designed and developed in order to provide functionality that is potentially reusable for future systems. Therefore, component-based software systems are built as an assembly of components already prepared for integration. The rapid assembly reduces implementation costs and potentially ensures high system-wide quality. Besides this, it enables easy system reconfigurations by substitution of individual components, which leads to high maintainability, scalability and evolvability. Finally, CBSE ensures sustainability and consistency of a global system architecture, where components can be considered as interchangeable blocks operating and communicating according to well-defined architectural rules.

However, it should be emphasized that the aforementioned benefits of

CBSE are achievable only under the assumption that the system development process fully complies with the CBSE standards. This compliance imposes certain technological overhead. Despite of the ongoing debates on industrial applicability of the CBSE technologies, evidence from various successful projects gradually becomes available [70, 62, 41, 90].

Moreover, in a nutshell, CBSE extends the discussed MDA principles with the concept of Predictable Assembly (PA) [50]. *Predictable assembly* allows an architect assembling a system (out of individual components) with predictable functional and extra-functional properties, e.g. performance. PA enhances the MDA approach with techniques for design-time model-based analysis of the quality attributes of systems composed from independently-developed arbitrary components. To enable the predictable assembly, a component provider should supply specifications of different attributes of components as a set of models. At the component assembly time, once the attributes of individual components are available, it should be possible to reason about the quality attributes of the complete assembly.

In general, the actual behaviour and resource consumption of a component-based assembly can be determined from the following data: (a) structure of the component assembly, (b) deployment of the components on hardware nodes, and (c) properties of the individual components related to behaviour and resource consumption. Taking a process view, design-time predictions on system performance require the following four steps. First, find and express the performance properties of individual components. Second, identify the component-assembly structure and mapping scheme on the hardware platform. Third, synthesize the properties of individual components based on the defined semantics, assembly structure and mapping scheme. Finaly, analyze the synthesized data and reason about the performance properties of the architecture.

A number of component-based technologies exists in the domain of resource-constrained systems that support some of the previously indicated process steps. These technologies include: ROBOCOP [56], PECOS [34], PECT [50], Koala [100], Rubus [92] and SaveCCM [46]. A detailed analysis of these technologies is presented in Chapter 3 of this thesis. Most of them address extra-functional requirements, which is crucial for safety- and time-critical systems. These technologies have the following common features that enable development and assessment of performance-critical systems: (a) lightweight deployment infrastructure that reduces technology processing and memory use overhead, (b) well-defined models for specification of components and their quality attributes, and (c) tools or methods for testing the quality attributes of the resulting component-based systems.

However, these CBSE technologies do not provide full-fledged support for design-time modeling and assessment of performance properties. The needed support should provide the following aspects:

- *modeling syntax* for specification of various performance properties of individual components;
- well-defined *semantics and rules for synthesis* of these models at the component-composition design phase;
- *reasoning framework* allowing to extract and analyze the processed models with respect to performance;
- *supporting software tools* enabling automation in the design and analysis phase.

There are also specific challenges in enabling the performance predictions. Firstly, due to the time-to-market pressure, an architect should be able to spend relatively *low effort* for designing an architectural alternative and predict its performance properties. Therefore, an ideal design method should provide facilities for *rapid* prototyping, design and analysis of architectures. At this point, the high accuracy of predictions is not the main target. Instead, an architect obtains course-grained measures and some clue on the consequences of the design decisions made.

Secondly, an architect needs the *back-tracing means* to map the prediction results onto the architecture and to understand how the design decisions influence the performance values obtained. Therefore, a sufficient level of analysis support is needed that may include visualization and guideline-enabling instrumentation. The result of this analysis can be specific improvements introduced to the architecture. Here, an architect should be able to easily change the architectural elements and re-evaluate the new alternatives. Such iterative design and comparison is called an *architecture optimization* and is of vital importance for development of complex real-time systems, where it is not possible to take all performance nuances of a system in one iteration.

Thirdly, the challenge is how to provide an architect the possibility to evaluate real-time systems built on *arbitrary hardware platforms*. For realizing this, a method should provide modeling facilities for various hardware configurations, as well as support for exploring different scheduling algorithms, caching strategies, memory- and disc-access algorithms.

Last but not least, from the software point of view, a method should be flexible and scalable enough to allow modeling and analysis of real-time systems built in an *arbitrary architectural style*. Depending on the requirements, a software architecture may deploy one of the following styles: blackboard, pipes-and-filters, publish-subscribe. The styles differ in threading, data synchronization, data storage and communication principles. As a consequence, an ideal method should provide the facilities for modeling and analysis of these variations in system aspects and, thereby support different architectural styles.

1.2 Problem Statement and Research Questions

In the previous section, we have shown that real-time systems address many societal needs and become increasingly important in our daily life. However, the development of real-time systems is a challenging task, due to the pervasive nature of their most critical property: performance. Performance quality attributes must satisfy the requirements in all foreseen situations. Moreover, the performance values should be predicted and assessed already at the early design phases, since performance faults in a realized system lead to costly re-design cycles or product failures.

Based on the above discussion, we formulate the *general problem statement* that is central to this thesis as:

> How can performance properties of system architectures be evaluated and improved during early stages of system development?

In order to refine the general problem statement, we narrow our focus from different prospectives. Firstly, we have the opinion that the CBSE principles reduce development time and cost, so that we concentrate on obtaining performance values of systems built according to the CBSE standards. In other words, we aim at providing support for systems *composed out of a set of third-party components*. We assume that a source code and executables of the individual components are not available to an architect, thus, he should be able to evaluate an architectural alternative without even buying the components but only based on the provided component models.

Secondly, from the whole spectrum of performance properties, we aim at the *most critical* ones: (a) task latencies, (b) usage of processing power, memory and network bandwidth, and (c) performance bottlenecks in a system behaviour. The bottlenecks are derivatives from the above-mentioned properties, however, they are important in the understanding of problem points in the architecture, e.g. insufficient capacity of hardware resources, task blocking issues or high-load picks on specific hardware blocks.

Thirdly, with respect to the amount of efforts, we focus on the problem of *rapid* performance predictions requiring relatively low efforts in modeling, reasoning and analysis. This is due to the fact that we would like to support modeling and analysis of *multiple* architectural alternatives, consecutively followed by a comparison of those alternatives. For an architect considering multiple alternatives, it is important to analyze their properties in a rapid and cost-effective way.

Taking into account these objectives, we specify our *refined problem statement* as:

How can the detailed task behaviour, and hardware resource-usage properties of a component-based system be rapidly predicted, based on the properties of individual components?

Research Questions

Decomposing the refined problem statement, we formulate a number of research questions to be addressed in this thesis as follows.

RQ1: How should behaviour- and performance properties of individual components be specified in order to enable automated composition of these properties into an analyzable model of a complete system?

RQ2: How to combine the models of individual components into the model of a complete system in an automated way, such that the resulting system model can be analyzed against the performance properties?

RQ3: How can architectural alternatives be compared and optimized with respect to multiple quality attributes?

RQ4: How can the assessment process of performance attributes be accelerated without a substantial reduction of the prediction accuracy?

1.3 Research Method

The research method types can be classified in deductive and inductive ones. The deductive type, also referred to as *top-down* approach, begins with establishing a theory about the topic of interest, proceeds through hypothesis and completes with validation of that theory. The inductive method, also called *bottom-up* approach, starts with observations of an existing situation/problem and ends up with a theory based on the observations and experiments made.

In our research, we have mostly applied the inductive approach, while enhancing it with the elements of deductive thinking. The reason for applying the inductive approach as a basis was that we aimed at finding a methodology that can be applicable in industrial environment. The drawback of this is that we sacrificed the "mathematical beauty" in favor of industrial applicability. Fig. 1.1 visualizes our research strategy. We have started with observation of current industrial problems in the domain of real-time systems. These observations have helped us to understand the down-to-earth needs and the constraints that the architects experience when developing time-critical systems. The analysis of the observations have led us to the ideas on how to address these needs and constraints. The exploration and enhancement of these ideas have resulted in creating a methodology, that we present in this thesis as the so-called DeepCompass framework. In order to validate the methodology, we

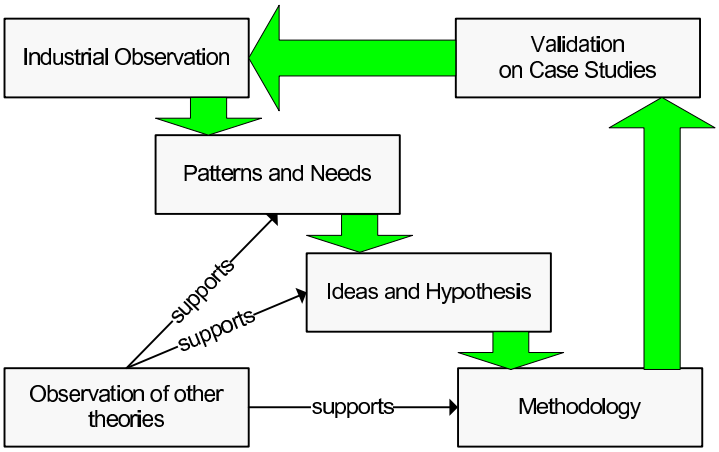

Figure 1.1: *Structure of our research method.*

have performed a number of case studies from the industrial environment. We have used the results of these case studies for further iterations on the methodology.

It is important to mention that we have also employed the observations of the state-of-the-art methodologies for performance analysis, while identifying the needs, generating ideas and establishing the methodology.

1.4 Major Contributions

The major contributions of this thesis can be classified into three categories: (a) DeepCompass framework, (b) scenario-based performance analysis method and (c) empirical survey on scenario-based performance analysis. The following paragraphs summarize these contributions.

The *DeepCompass framework* defines a design-for-performance development process that guides an architect through iterative design cycles. Besides the modeling and design steps, each cycle iteration incorporates such important steps as early performance analysis and architecture optimization. For each step, the framework defines the activities to be performed by an architect, the deliverables, as well as the interdependencies between the steps. The framework is supported by a CARAT software toolkit that provides an architect the graphical design tools, computationally complex algorithms, and verification/visualization means. From the business point of view, the major benefit of the framework is that it enables *rapid* design and performance analysis of component-based systems, purely based on models of independently developed individual components. The use of models allows to design and analyze

a component-based system without even buying the constituent components.

A *scenario-based performance analysis method* is the core of the DeepCompass framework. The method enables predictions of detailed performance properties (task interleaving and blocking, latencies, and hardware-resource utilization) already at the *early design phases*. These benefits are achieved by the following innovations: (a) composable models of individual components, (b) model-synthesis algorithms able to generate a system model out of component models, (c) simulation algorithms applied to the system model, and (d) scenario-based modeling of interactions of the system with its environment. Assessment of performance properties only for a set of critical scenarios substantially reduces the assessment time and efforts. Another important contribution of the method is that it enables automated synthesis of individual component models into a system-wide model for any proper composition of an arbitrary set of components. The broad applicability of the method for stream-based and control-based systems is supported by different modeling primitives and algorithms addressing various types of communication/architectural styles and different hardware platforms with heterogeneous processors. We have validated the method by performing three case studies on an MPEG-4 decoder, a Car Radio Navigation system and a JPEG application.

A *survey on usage of scenario-based analysis methods* is carried out among experienced architects from different application domains. The survey results in empirical data and practical knowledge on performance analysis and architecture evaluation using scenarios. The survey reveals that scenarios are widely used in industry for analysis of real-time systems. All interviewed architects mention that they use scenarios in practice. However, the interviews also show that scenarios are not a "silver bullet" solution in architecture assessment, because they have clear advantages and drawbacks in comparison to the opposite paradigm: formal methods. The survey delivers guidelines and recommendations for identification and deployment of scenarios in performance analysis.

1.5 Thesis Outline

This section gives an outline of the chapters in this thesis and summarizes the contributions of individual chapters. The logical structure of the thesis is depicted in Fig. 1.2 and is described as follows. Chapters 1-3 aim at preparing the reader to the core of the thesis. Here we define our problem statement and accompany it with the domain analysis and with the detailed problem analysis. Chapters 4-6 describe our design and analysis methodology, including the overview of the DeepCompass framework, followed by the detailed specification of the comprising scenario-based performance-prediction approach and the architecture optimization method. An experienced reader may skip Chapters 2-3

and proceed from Chapter 1 straight to the main Chapters 4-6. Chapters 7-9 deal with validation of the presented framework.

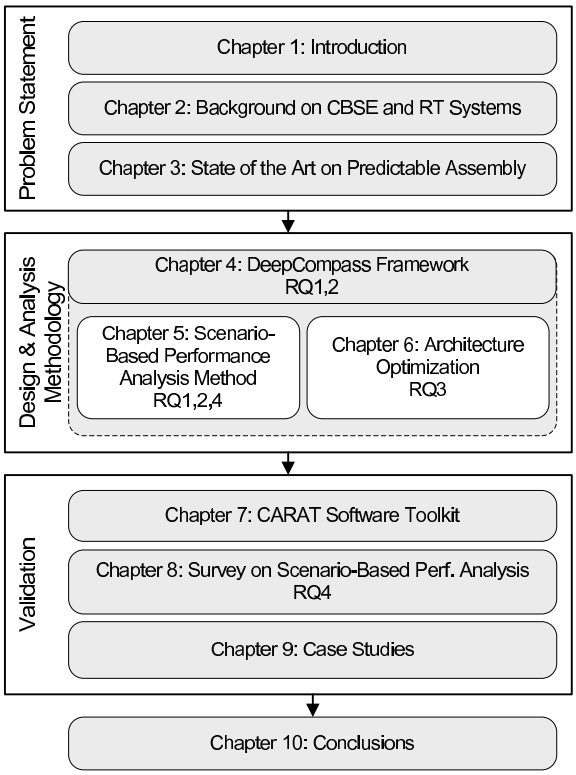

Figure 1.2: *Logical structure of the thesis and labeling of research questions.*

The remainder of this section summarizes the content of individual chapters and indicates the relation between the thesis chapters and our publications.

Chapter 2. In this chapter, we provide the reader with the definitions and technologies used in this thesis. We present the CBSE concepts, and outline the ROBOCOP software-based technology while discussing the types of components used in the industry. Furthermore, we introduce definitions for real-time systems and related performance properties.

Chapter 3. This chapter presents the state-of-the-art methods addressing performance analysis. We classify the performance analysis methods and provide a number of examples for each class. Besides, we give an overview on how the performance predictions are handled within the component-based system community. We explore the most mature component-based approaches that enable the design-time model-based performance predictions and highlight the

advantages and limitations of each approach. This chapter is somewhat extensive due to co-authored tutorials at the IEEE ICCE 2006 Conference and IEEE Euromicro 2006 Conference, which presented the ROBOCOP architecture and the newly developed framework from the author of this thesis.

Chapter 4. This chapter opens up the core part of the thesis. It describes the proposed performance analysis framework from the development process point-of-view. The chapter outlines how the framework supports an architect in the phases of selection of individual components, design of architectural alternatives, performance analysis and optimization of the alternatives. The framework achievements were initially published at the IEEE Euromicro SEAA 2006 Conference [13] of the IEEE Computer Society, and at the CBSE 2006 Conference [18] of the ACM and published by Springer. Moreover, the framework description was also accepted as a journal publication in IEEE Transactions on Software Engineering [15], but the publication is still under revision.

Chapter 5. We present the analysis method which serves as a framework core from the technical point of view. The method is based on defining critical execution scenarios of a system and analyzing the performance for these specific scenarios. The method features (a) modeling of individual software and hardware components at a high abstraction level, (b) specification of architectural models containing scenarios, which define stimuli that trigger task executions within a system, (c) automated synthesis of individual component models and architectural models into an executable system model, representing a specification of running tasks in a system, and (d) simulation of the tasks resulting in predicted performance properties. In this chapter, we provide all the low-level details of the method in order to make the results repeatable for other explorations. The method was published earlier at the IEEE Euromicro SEAA 2004 Conference [17], as well as at the IEEE Euromicro SEAA 2005 Conference [16] and a related Euromicro workshop on Dependable Software Intensive Embedded Systems [19].

Chapter 6. We extend the DeepCompass framework with architecture optimization functionality. This chapter outlines existing methods for optimization of architectures and describes an abstract method, which we derived from the existing methods and our own experience. Later in the chapter, we tailor the abstract method to the DeepCompass framework. The optimization method was published as a book chapter in a book from Grunske *et al.* entitled "*Architecting Dependable Systems IV*" from Springer [45].

Chapter 7. This chapter describes a set of software tools integrated in the so-called CARAT toolkit. CARAT was developed by the author in order to support and automate all the iterative phases of the DeepCompass Framework. The toolkit includes the following modules: Repository, Graphical Designer, Preprocessor, Simulator and Visualizer. In this chapter, we describe the toolkit architecture and show how each module supports different phases in the frame-

work. The CARAT toolkit was initially published at the IEEE ICSEA 2006 Conference [11] and at the IEEE/ACM DATE 2007 Conference [14].

Chapter 8. This chapter provides the results of our empirical study on the usage of scenarios for design-time performance analysis. This chapter reflects the hands-on knowledge of experienced industrial architects dealing with scenario-based performance analysis. The survey addresses the *pros* and *cons* of using scenarios and justifies the feasibility of applying scenarios for performance analysis at the early phases of the architect's projects. Also, this chapter provides indications that usage of only critical scenarios accelerates performance analysis without substantial reduction in prediction accuracy.

Chapter 9. In order to validate the feasibility and accuracy of our framework, we carry out three case studies on an MPEG-4 decoder, a Car Radio Navigation system and a JPEG application. The case studies include the following steps: modeling of the software and hardware components; design of software and hardware architectural alternatives; and performance analysis of these alternatives followed by identification of an optimal architecture. Besides this, we implement and profile the applications in order to obtain actual performance values and compare them with the predicted ones. The studies help to reveal the limitations of the framework and prediction accuracy problems. The findings from each case study serve as an input for iterative improvement of the framework. Parts of this chapter were published earlier at the ACM WOSP 2007 Workshop [12], at the ACM CBSE 2006 Conference [18], and at the SPIE VCIP 2005 Conference [20].

Chapter 10. This chapter concludes the thesis, discusses the limitations and benefits of the presented framework and ends with opportunities for future work.

Chapter 2

Background on CBSE and RT Systems

2.1 Introduction

Nowadays component-based software technology is applied increasingly for time-critical system development. As a result, the problem of early assessment of performance properties becomes important and vital for the success of such systems. In this chapter, we explain the main concepts used in the domain of component-based software systems. The concepts are grouped in the following topics: (a) component-based software engineering, and (b) real-time systems and performance properties. We introduce basic definitions and concepts used throughout the thesis.

Section 2.2.1 describes main advances and challenges in the CBSE discipline, in particular with relation to performance-critical system development. Besides this, Section 2.2.2 differentiates between passive and active components used in different application domains. Section 2.2.3 specifies the ROBOCOP component model as an example of a component-based technology. Section 2.3 describes the types of performance properties and related system issues such as events, logical execution tasks, and hardware resources. Section 2.4 concludes the chapter.

2.2 Component-Based Software Engineering

This section discusses the concepts of the domain of component-based software engineering. We give the definition of a software component, explain the

component modeling, composition and infrastructure issues. Then we differentiate between passive and active component types and provide the ROBOCOP example of a component-based framework.

2.2.1 Component-Based Architecture Definitions

The CBSE discipline sometimes exposes contradictory or confusing definitions of basic terms. For instance, a *component model* can apply to an individual component as well as a complete component-based architecture. Let us now further clarify these terms.

Software Component: Szyperski [93] has defined a software component as *a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* According to Szyperski extended definition, a software component is:

- A subject for multiple use. A software component should be designed and implemented such that its functionality can be reused in many different systems.
- An externally stateless entity. A component does not expose his execution state to a system and can be bound, started and stopped at any moment of a system lifecycle.
- Composable with other components. A component provides well-specified interfaces, by which it can be bound to its neighboring components.
- An encapsulated entity, i.e. a component internal implementation cannot be explored through its interfaces.
- A unit of independent deployment. All component dependencies on external resources are clearly specified and it can be substituted by some other component.

Since the component's internal implementation is encapsulated from the outside world, a component exposes its functionality and connectivity specification via its interfaces. A *component interface* is a set of named operations with specified signatures, that can be invoked by other components. In other words, a component offers access to its functionality via its interfaces.

A component may have two interface types: *provided* and *required* interfaces (see Fig. 2.1). Whereas a provided interface specifies the functionality that a component offers to the environment, a required interface specifies a component's requirements to the environment that have to be satisfied for proper operation. More specifically, required interfaces are ports through which

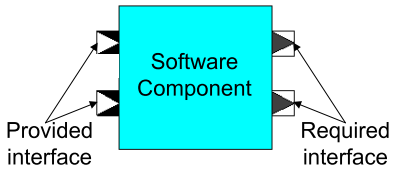

Figure 2.1: *Software component with provided and required interfaces.*

a component can invoke operations provided by other component interfaces. At component deployment, a required interface can be bound to a provided interface of another component.

Component Model. This describes various constraints for component development. These constraints include requirements for: (a) the component development and deployment process, (b) component implementation issues (programming languages, interfaces implementation), (c) specification of models of component properties including behaviour, resource use, context-dependencies, etc. Besides this, a component model serves as a set of guidelines for an architect, specifying the rules for composing individual components into an assembly.

Component Composition. A composition of components, sometimes called component assembly, is a set of instantiated components and bindings (connections) between their respective provided and required interfaces. An architect selects and composes components in order to satisfy system requirements. The rules for creating a component composition are defined by the component model and specifies how components should be bound, while the rules of the component framework describe how a component should be integrated with an operating system and run-time framework.

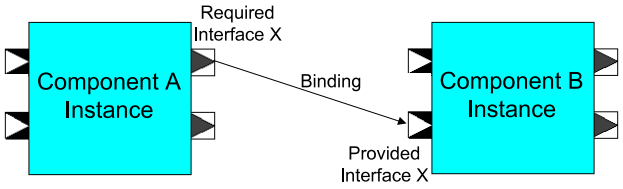

Figure 2.2: *Composition of two component instances.*

The deployment issues include component creation and deletion, as well as facilities for control and communication from an operating system. In terms of programming, a component composition is glue code that instantiates, binds

components and assigns them to specific hardware processing nodes. Fig. 2.2 represents a simple example of a component composition. An instance of Component A is bound to an instance of Component B via their interface of the same type X. This composition allows Component A to invoke operations of Component B via Interface X.

Component Framework. This is a middleware layer built on top of an operating system. The goal of a component framework is to enable proper component creation, binding, deployment and operation. Components may provide interfaces to a framework to enable access to lower architectural layers. For example, components may have a required interface to be bound to a framework in order access to some system functionality, like a system clock. A component framework typically supports one single component architecture.

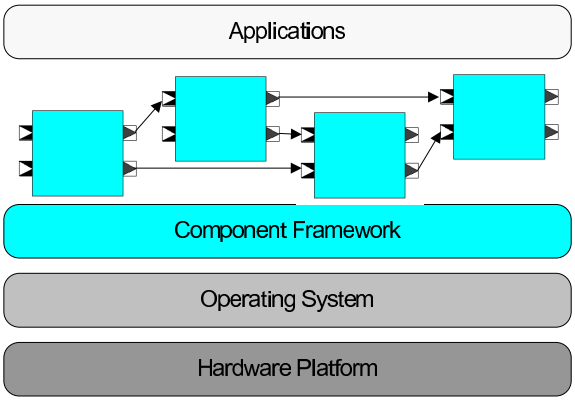

Figure 2.3: *General system architecture with component-based software framework.*

Fig. 2.3 shows an architecture of a general component-based software system. The lower system layers contain the hardware platform and operating system. The component framework provides an infrastructure for component registration, creation, binding and execution. Besides this, the framework enables component deployment on the hardware platform. The application layer contains various end-user applications that use the underlying components for their operation.

2.2.2 Passive and Active Components

Different domains use specific types of architectural styles. For example, control systems often employ synchronous communication between *passive components*, while multimedia systems are based on pipes-and-filters architectures

with *active components* and buffers in between them. In the remainder of this subsection, we explain the differences between passive and active software components.

There are a number of interaction (communication) styles that components may employ: synchronous method call, remote procedure invocation, message passing and buffer-based pipelining. Depending on the interaction style required, either active or passive components are used.

We call a component *active* when it has at least one process (thread of control) that is started and executed within the component's boundaries. This process normally executes a `while-do` loop. Within this loop, the process reads data from input ports, executes operations and writes data to output ports. The ports of communicating components are connected via a buffer or a channel, passing the data from an output port to an input port. Active components may communicate both in synchronous and asynchronous ways. Active components are extensively used in dataflow-oriented multimedia applications.

A *passive* component does not activate and run any processes inside its boundary. A passive component provides access to its implemented operations for processes (thread of controls), which are created outside the component boundary. Hence, this represents static code waiting to be invoked for execution. Passive components communicate in a synchronous way: an operation of one component invokes an operation of another component via a specified interface and waits for the return of the thread of control. Passive components are widely used in control systems. Note that *mixed* components integrating both concepts are also possible.

In most of the current architectures both active and passive types of software entities are utilized in mixed form. Passive entities implement control operations and active entities realize dataflow streaming. For this reason, an important requirement for our performance-prediction method is to support modeling and analysis of all passive, active and mixed component types.

2.2.3 ROBOCOP Component-Based Architecture

We have adopted the ROBOCOP component-based architecture [56] for conducting our research on performance analysis. ROBOCOP stands for Robust Open Component Based Software Architecture. This architecture was developed for middleware in consumer devices, with an emphasis on robustness and reliability. ROBOCOP is inspired by CORBA [75] and Koala [100], but provides more efficient support for realization of real-time and performance constraints via modeling techniques.

A ROBOCOP component is a set of possibly related models, as depicted in Fig. 2.4(a). Each individual model provides specific information about the component. Models can be represented in readable form (e.g. documentation

or XML-files), or in binary code. One of the model types is the *executable model* that contains an executable component. Other examples are *security model* and *reliability model*. The set of models is open and a third-party component provider may add its own model type. The model set should be specified and packaged at the component development phase. These models are an indispensable part of the component description and facilitate component trading and analysis.

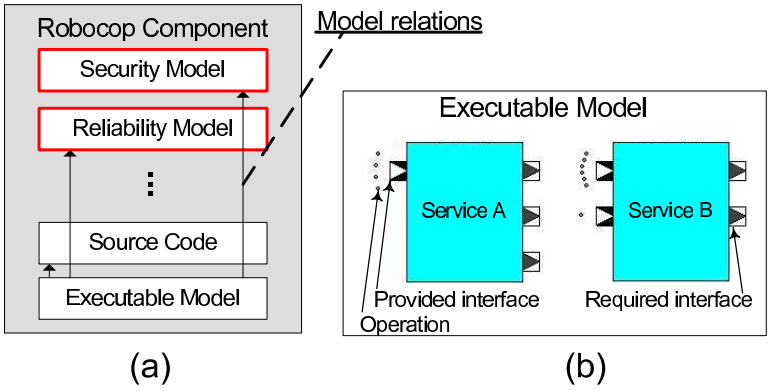

Figure 2.4: *(a) ROBOCOP component model, (b) ROBOCOP executable model.*

A ROBOCOP executable model specifies a set of executable entities called *services*. Services are instantiated at run-time. The resulting entity is called *service instance*, which is a ROBOCOP equivalent of an *object* in Object-Oriented Programming (OOP). An executable component offers functionality through a set of services (see Fig. 2.4(b)). Services are static entities, which are the ROBOCOP equivalents of conventional CBSE components described in Subsection 2.2.1, or of *public classes* in OOP. A ROBOCOP service defines a set of interfaces. The ROBOCOP model distinguishes *provided* interfaces and *required* interfaces. An interface is a set of operation signatures. The binding between service instances in the application is implemented via a pair of provided-required interfaces. Comparing conventional CBSE and ROBOCOP definitions, a ROBOCOP *component* is a CBSE package or unit of trading, while a ROBOCOP *service* is what CBSE defines as a component.

In ROBOCOP, as well as in other component models, service interfaces and service implementations are separated to support *plug-in compatibility*. This allows different services, implementing the same interfaces to replace each other. As a consequence, the actual implementations to which a service is bound do not need to be known at the time of designing a service. This implies that resource consumption of a service cannot be completely determined for

an operation, until an application defines a specific binding of the required interfaces of service instances to provided interfaces.

2.3 Real-Time Systems and Performance Properties

This section gives an introduction to the domain of time-critical systems. It defines different types of time-critical systems and shows the kind of requirements that such systems should satisfy. Moreover, this section explains a number of important performance definitions, like latency, throughput and logical tasks.

2.3.1 Real-Time Systems

Due to the increasing popularity of time-critical embedded systems, a number of definitions and aspect of a real-time system have been defined. The most commonly used definition has been given by Timmerman in [97], where a Real-Time (RT) system is defined as a *system that responds in a (timely) predictable way to unpredictable external stimuli arrivals.* An RT system should fulfil the following types of requirements under extreme load conditions.

- **Timeliness.** A system should meet its deadlines, i.e. it has to complete certain tasks within the time boundaries specified by real-time requirements.
- **Predictability.** A real-time system has to react to all possible events in a predictable timely way.
- **Allowing simultaneous processing.** If multiple events/stimuli occur simultaneously, then all deadlines should be still met.

A system is said to be real-time, if the functional value of an operation depends not only upon its logical correctness, but also upon the time in which it is performed. The classical conception is that in a hard or immediate real-time system, the completion of an operation after its deadline is considered useless - ultimately, this may lead to a critical failure of the complete system.

However, in current systems not all deadlines are so critical. Depending on the deadline failure criticality, the following types of RT systems exist.

- **Hard real-time.** Missing a deadline results in catastrophic failure of a system. In hard RT systems, the cost of missing a deadline is infinitely high and no lateness is accepted under any circumstances. Examples are aircraft, defense or automotive control systems.

- **Firm real-time.** Missing a deadline leads to unacceptable quality reduction. The value of the function completed after a deadline is null, however, it does not entail any catastrophic consequences. For example,

a video application that completes a frame decoding after its deadline cannot render the frame on a monitor and needs to skip it. The skipping of a frame is not catastrophic, but reduces the perceived video quality.

- **Soft real-time.** Missing a deadline leads to acceptable quality reduction. Deadlines may be missed and the system can be recovered, leading only to degradation in functionality value. In soft RT systems, while a certain amount of system latency is acceptable, the event must still be reacted to within a deterministic period of time. One example is an online transaction system, where substantial response delay reduces the customer perceived quality of a system and may lead to reduction of the number of clients and transactions.

This categorization of RT systems appears due to the cost vs. performance trade-off. It is always hard and costly to build a system that meets deadlines under all circumstances. Therefore, in order to reduce production costs and development efforts, engineers relax the system requirements wherever possible. Soft real-time systems are a good example of such relaxation.

As can be noticed, the most important performance quality attribute of an RT system is latency. However, there exist a number of other performance properties that may be specified in RT system requirements, like throughput, task execution time and processor usage. The following subsection explains these concepts from performance engineering in detail.

2.3.2 Performance Properties

Throughout this thesis, we use the notion of performance properties and performance quality attributes equivalently. In general, performance is a global term involving various system aspects, like responsiveness, throughput and resource occupation. A performance property can be defined as a metric of a system that shows how fast a system executes certain functionality. We group performance properties into application-specific and system-specific types. The following paragraphs explains both types in detail.

A. Application-Specific Performance Properties

Application-specific properties include throughput, task execution time, latency and delay. Generally speaking, *throughput* is the amount of work that an application can perform in a given time period. In this context, the word "work" may imply: (a) the amount of user or environmental requests fulfilled by an application, (b) the amount of internal jobs (task instances) executed, or (c) the amount of data transferred from one place to another.

Originally, throughput has been used for measuring performance of distributed applications. In this context, throughput measures the number of bits that an application is able to transfer per unit of time, e.g. in a second.

An individual node throughput is the amount of data that the node is able to receive, process and send in a second. At present, throughput is an important property in many application domains, like multimedia and control systems. For instance, in video streaming applications, throughput is measured by the amount of video frames per second that an application can receive, decode and render on a monitor. In control applications, throughput is measured by the number of environmental and platform events that an application is able to fully process in a second.

Latency is another important performance property. Throughout the thesis, we consider that latency and response time are equivalent parameters. In a software application context, a latency is the time period between arrival of a stimulus triggering some application activity and the completion of this activity, or the corresponding output stimulus triggering the result delivery.

Latencies and deadlines are two sides of the same coin. In most cases, performance requirements of real-time systems are given in terms of deadlines specifying maximum latency allowed for a certain activity to complete. Therefore, latency prediction is an inevitable challenge for architects of time-critical systems.

Computing latency for complex systems is a non-trivial task. There are many factors influencing the latency value: availability of processing and networking hardware resources, memory access, task interleaving and scheduling peculiarities. We address these challenges in Chapter 3. Here, we briefly explain the constituent parts of the latency computation.

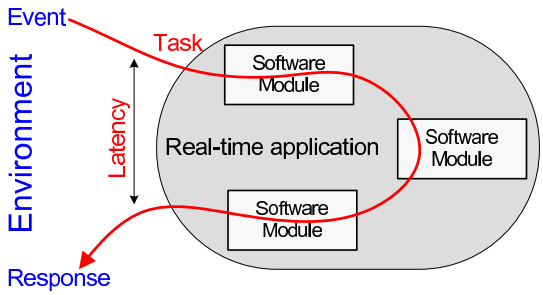

Figure 2.5: *Latency of a task executed by a simple application.*

Fig. 2.5 depicts a real-time application receiving an event from the environment. An event is addressed by a real-time requirement, specifying a deadline for the event response. An environment can represent an end-user, sensors, or other systems. The event triggers a logical-task instance, which consists of a number of operations executed by certain software modules. Once the operations complete execution, a response is delivered to the environment. The time difference between the event and response is the task latency, which should be

checked against the deadline.

However, an application may receive and process a number of such events in parallel. Here, an application should execute a number of corresponding tasks simultaneously. Moreover, an application may be deployed and executed in a distributed manner on a number of hardware processing nodes. In this case, latency computation of each individual task is not so straightforward and depends on the deployment of software modules on processing nodes and task-scheduling policies of the processors. An abstract example of this situation is shown on Fig. 2.6.

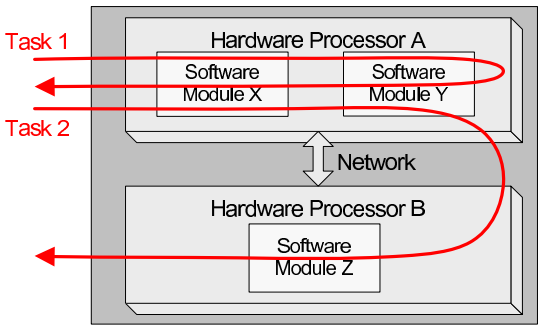

Figure 2.6: *Tasks executed in parallel by a distributed application.*

An application is deployed on two Processors A and B, connected to the same network. Software Modules X and Y are deployed on Processor A, while a Software Module Z is deployed on Processor B. Task 1 and Task 2 are executed in parallel. Task 1 is passing through Modules X and Y, while Task 2 is passing through all three software modules. Both tasks are executing on Processor A in parallel, which means that the Processor A schedules their execution. This implies that the low-priority Task 2 needs to wait until the other high-priority Task 1 is completed. This time of waiting is called blocking time. The latency computation of Task 2 needs to incorporate this blocking time. Moreover, Task 2 experiences some additional latency when the task-related data is passed from Processor A to Processor B via the Network. This latency is called communication time. If the network is heavily loaded, it may also introduce blocking time for Task 2. In general, a task latency may be computed by the following equation:

$$T = \sum_{i=1}^{n_1} Dp_i + \sum_{i=1}^{n_1} Bp_i + \sum_{j=1}^{n_2} Dc_j + \sum_{j=1}^{n_2} Bc_j, \tag{2.1}$$

where Dp_i is the processing time delay introduced by a processor i during the task execution, Bp_i is the blocking time introduced by a processor i, Dc_j is

the communication time delay introduced by a network j for the task data transfer, and Bc_j is the blocking time introduced by a network j, respectively.

In case a task accesses an external memory or storage device, the Equation (2.1) needs to be extended with respective access and blocking time delays.

The processing and communication time delays can be assessed analytically, without running or simulating a system. The processing time delay $Dp(t)$ of a task t executed on a processor p can be computed by:

$$D_p(t) = C_t/F_p, \tag{2.2}$$

where C_t denotes the number of processing cycles required to execute the task t, and F_p is the processing frequency of the processor p measured in terms of cycles per second.

The communication time delay $Dc_q(t)$ of task t introduced by a network q can be calculated by:

$$Dc_q(t) = S_t/B_q, \tag{2.3}$$

where S_t is the amount of bits that the task t sends through the network q, and B_q is the bandwidth of the network q, measured in terms of bits per second.

This coarse-grained latency definition does not include other influencing hardware factors. For example, disc I/O, memory access, processor cache-misses, and architectural mismatches may increase the task latency.

The communication and processing blocking-times may vary during a system run, because they depend on a run-time system-specific performance properties, such as processor and network loads. Therefore, simulation is a suitable method for obtaining these metrics. Analytical approaches allow obtaining only worst-, average- and best-case values for these metrics. In the remainder of this section, we address system-specific performance properties.

B. System-Specific Performance Properties

System-specific properties are measures taken for the underlying system platform. These properties are an essential constituent for the computation of application-specific properties, which need to be investigated during performance analysis of an application.

System-specific performance properties can be divided into two types: manufacturer-defined and run-time properties. The manufacture-defined properties are given in the hardware specification and are constant over the system execution, e.g. network bandwidth, while the run-time properties show the current load of the hardware resources and vary over the system execution. Major system-specific properties can be categorized by the following types of hardware resources.

- **Processor.** A manufacturer-defined performance property is a *processor frequency* specifying the number of cycles per second executed by the processor. An important run-time property is *processor usage* showing the utilization of the processing power by one application task or by the whole system. In this thesis, we use the notion of usage and load equivalently. A processor load of 100% means that applications are utilizing the processor to its maximum capacity. The processor usage can be momentary (measured at that precise moment of execution), average-case and worst-case. The worst-case processor usage is a maximum value of processor load measured during a system execution.

- **Network.** A manufacturer-defined property is a *bandwidth* describing the number of bits per second that the network can transfer. A network run-time property is the *network load* representing the network bandwidth utilization in percentage by one application task or by the whole system. Similarly, the network load can be classified as momentary, average-case and worst-case load.

- **Memory.** A manufacturer-defined property is a *memory capacity* specifying the number of bytes that can be stored into a memory. *Memory usage* is the memory capacity utilization in percentage by system applications.

The manufacturer-defined properties serve as an input for performance analysis, while the run-time properties represent the output of the analysis.

Computation of the run-time performance properties can be performed both at the design-time and execution-time. The design-time methods for computing include static analysis and simulation. The execution-time analysis can be performed by system profiling and monitoring.

2.4 Summary of the Background

In this chapter, we have provided background knowledge on component-based software engineering and performance properties. A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A component interface is a set of named operations with specified signatures, that can be invoked by other bound components. We have also learned that a software component can be deployed independently and component-based system is composed of individual components developed by a third party. The principal requirement for each individual component is to be compliant with the model and framework of the component-based architecture used for development.

With respect to the domain of time-critical systems, we have distinguished a classification into the hard-, firm- and soft real-time systems, and also discussed main performance properties of such systems. These properties include latency, throughput and hardware resource usage. We have defined latency as a time period between the arrival of a stimulus triggering some application activity and the logical completion of this activity. In general, throughput is computed as the amount of work that an application can perform in a given time period. We have also outlined how these properties are computed for a system, where software components are mapped on a multi-processor platform.

In the next chapter, we discuss the state-of-the-art methods addressing the problem of design-time analysis of these performance properties. Besides this, we give insights on how the component-based development can help to properly handle the previously mentioned problem. We show that the concept of component's composability, as a main feature of CBSE, can also be used to realize composability of performance properties. In other words, we discuss how the system properties can be obtained by composing the properties of individual components.

Chapter 3

State of the Art on Predictable Assembly of Components

3.1 Introduction

The previous chapter outlines the main concepts of component-based software engineering and performance analysis. Performance is a challenging system quality to predict, measure and enforce. Performance properties depend on various factors, like environmental context, load profile, middleware, operating system, hardware platform and sharing of internal resources. Conquering this complexity of performance is a vital issue for development of systems with performance requirements. It causes project delays, budget increases and even project cancelations. A scalable and, at the same time, rigorous methodology is needed for allowing system developers to successfully handle the performance properties throughout the project phases.

In this chapter, we show that performance prediction can be facilitated by the concept of *predictable assembly*, introduced by the CBSE community. The idea of predictable assembly is to use the properties of individual components to reason about the quality attributes of a system composition. One of the main challenges in constructing and maintaining software is to express and verify quality attributes of applications. The concept of predictable assembly of components enables expressing and predicting quality attributes derived from the properties of the individual components. We present and compare

the state-of-the-art methods, enabling the predictable composition of components and their quality attributes. Our special focus is on performance-related compositional methods.

The chapter is structured as follows. Section 3.2 presents a classification of quality attributes with respect to their composability. Section 3.3 describes and compares the performance methods deploying the concept of predictable assembly. In Section 3.4, we provide a comparative analysis of these methods against multiple architectural criteria. Section 3.5 concludes the chapter.

3.2 Composability of Quality Attributes

Most component models define some form of component interface standard that enables the composition of individual component implementations. However, most of these models do not provide methods for the composition of the quality attributes of individual components. This reduces the value of the conventional component models for systems with extra-functional requirements on the quality attributes. However, the CBSE concept of composing systems from the individual components is also applicable to quality attributes. Namely, the task is to identify a system-wide quality attribute by composing the quality attributes of the individual components.

This composition task is not trivial because the diverse types of quality attributes do not have the same underlying conceptual features. Crnkovic [27] classified the types of extra-functional quality attributes with respect to their composability. Some system quality attributes can be derived directly from the component attributes, while others might require a complex calculation model, related to the component model and the system architecture. The following types of attributes/properties are distinguished related to composability:

- directly composable properties,
- architecture-related properties,
- derived properties,
- usage-dependent properties,
- environment-dependent properties.

This classification is non-orthogonal, hence the same property can belong to several types. The following paragraphs explain the classification in detail.

Directly composable property - a property of an assembly which is a function of, and only of, the same type of property of the components involved. An example of a property of this type is the *static memory size* of a component or an assembly, which is also known as the memory footprint. The simplest composition model is the calculation of the static memory of an assembly as the sum of the memories used by each component.

Architecture-related property - a property of an assembly which is a function of the same type of property of the components and of the software architecture. In this case, the assembly properties depend not only on the component properties but also on the architectural structure. Examples of such properties are *robustness*, *scalability* and *availability*. For example, the availability of a system can be improved by duplicating a server, executing the same software component.

Derived property - a property of an assembly which depends on several *different* properties of the components. The computation of these properties is a result of the composition of different component properties. An example of such a property in a real-time system is the *response time* that is a function of different component properties, such as *Worst Case Execution Time* (WCET), *period of execution* and *blocking time*. In order to identify this function, one needs to combine the above-mentioned real-time properties of the components together with particular system characteristics.

Usage-dependent property - a property of an assembly which is determined by its usage or workload profile. Such properties of an assembly depend not only on the internal properties of the components and their composition, but also on the particular use of the system. An example of this property type is *processor load*. The processor load depends both on (a) the processor usage during an execution of operations of individual components and (b) the frequency of invocations of these operations.

A particular problem with this type of property is the limited possibility of reusing measured and derived properties. If the usage profile is changed, the properties must be re-calculated or re-measured. An example of such property is *reliability* which is calculated or measured for particular usage profiles. Most of the performance-related quality attributes can be put in the category of usage-dependent properties. The *latency*, *throughput*, *processor- and network load* quality attributes depend on how the system is used, namely on the rates of events entering the system.

Environment-dependent property - a property which is determined by other properties and by the state of the system environment. The property depends not only on the usage profile, but also on the environment in which the system is executed. A value of such property can vary in different contexts (i.e. surrounding environment) in which the system is operating. An example of such a property is *safety*. It is obvious that in different circumstances, the same property may have different degrees of safety even for the same usage profile.

3.3 Predictable Assembly Methods

This subsection presents the state-of-the-art methods that enable design-time predictions of system-wide quality attributes, based on attributes of individual components. The description of each method is divided in two parts representing the following aspects: model specification (syntax), model transformation and interpretation (semantics). The *syntax* of a method defines a specification language, i.e. how the certain characteristics of an individual component or a complete system are described in a model. An example of modeling syntax is the UML-based logical, dynamic, development, use-case and deployment views. The *semantics* define how the models are composed or converted into some form, providing a meaning for system-wide performance analysis. Examples of semantics are simulation engines or model-solving algorithms. Such examples result in a system-wide model of real-time tasks, propagating through software components and even processing nodes.

In the following subsections, the state-of-the-art methods are discussed with the syntax and semantics as key views. Besides this, we outline benefits and drawbacks of each method. The majority of the presented methods have the following benefits: (1) low modeling effort, (2) automated synthesis of an assembly model, unless stated otherwise.

3.3.1 PECT

Prediction-Enabled Component Technology (PECT) is one of the first approaches to achieve predictable assembly from certifiable components. It is the approach taken by SEI and it is executed in the PACC research initiative in collaboration with ABB and Malardalen University [50].

Syntax

PECT uses a Construction and Composition Language (CCL) to produce specifications that contain structural, behavioural, and analysis-specific information about individual components and assemblies. The structural specification defines sink and source pins of a component, via which a component communicates with other components. CCL enables to specify component behaviour and performance properties of the operations provided by the component. The behaviour is specified by a reaction to the input signal arriving at the sink or source pins.

Semantics

The CCL specifications of individual components are translated to one or more reasoning frameworks that analyze and predict the runtime properties of assemblies. The translation is enabled by the CCL processors that auto-

mate the synthesis of the assembly models. PECT supports three reasoning frameworks. First, LambdaSS [49] predicts average latency of stochastic tasks managed by a sporadic server. Second, LambdaABA [49] predicts average latency in assemblies with periodic tasks. Third, ComFoRT [57] formally verifies temporal safety and liveness of the assembly.

PECT features the following advantages:

1. extensive tooling support by a set of graphical, modeling and computational software tools;
2. well-defined component model with reference implementations, which includes definition of all development phases, and run-time execution framework;
3. combination of both analytic and simulation analysis techniques;
4. support for both synchronous and asynchronous communication styles.

The technology has the following limitations:

1. absence of modeling support for heterogeneous multi-processor platforms;
2. low granularity of component behaviour modeling preventing the specification of conditional statements, loops, and parameter-dependent executions;
3. no explicit means for modeling of a system context/environment.

3.3.2 KLAPER

MOF/KLAPER [39] defines a model-driven transformation framework, centered around a kernel language KLAPER (Kernel LAnguage for Performance and Reliability Analysis) whose aim is to capture the relevant information for the analysis of non-functional attributes of component-based systems, with a focus on performance and reliability.

Syntax and Semantics

The KLAPER language enables transformations from a variety of design specification types into models for different types of performance/reliability analysis. As an input, KLAPER accepts software component models and hardware node models specified in UML or OWL-S [28] notations. From the input, KLAPER builds an intermediate model according to its transformation rules. The intermediate model represents an assembly model synthesized from models of individual components and the system design model. Various outputs are possible, each suited for a specific analysis type, e.g. Discrete Time Markov Processes (DTMP), EQN and Petri Nets.

KLAPER provides primitives to describe *services* (component operations) and their *behaviour*. The *workload* primitive allows to define the input events

or frequency of service executions, which completes the behaviour characterization of a system. The hardware nodes are modeled via the *resource* primitive, where the node's capacity, scheduling policy and the executed services can be specified.

KLAPER provides the following benefits:

1. modeling support for heterogeneous multi-processor platforms;
2. tools for transformation of the UML-based component and assembly specifications into the models for various types of analysis;
3. support for multiple types of performance analysis techniques;
4. support of both synchronous and asynchronous communication styles;
5. detailed parameter-dependent behaviour modeling of a component;
6. facilities for modeling of a workload on a system;
7. support for modeling and analysis of communication latencies and network load.

The limitations of KLAPER include the following:

1. software tools for modeling and computations are still not available;
2. different variations in the usage of an assembly are not explicitly modeled;
3. to our knowledge, validating case-studies for industrial systems are not reported.

3.3.3 Palladio Component Model

The Palladio component model [6] supports predictions of extra-functional properties of CBSE systems. Its meta-model enables characterizing parametric context dependencies on system resources. The performance analysis is simulation-based, where parametric dependencies are taken into account.

Syntax

Palladio enables specification of a component behaviour and performance properties via a set of the following constructs. A ServiceEffectSpecification (SEFF) describes how the provided service calls the required services of the component. Internal behaviour of the Palladio services is modeled via the ResourceDemandingSEFF construct that contains data on the hardware resource usage, transition probabilities, loop iteration numbers, and parameter dependencies. The hardware resource usage of a service is modeled via the ResourceDemandingActions construct that specifies loads on the resources. The variations in behaviour of services and resource usage are modeled via random variables and Probability Mass Functions.

In order to specify the system-wide behaviour, Palladio introduces a UsageModel construct. The UsageModel consists of a number of UsageScenarios,

which in turn consist of one ScenarioBehaviour and one Workload meaning that the ScenarioBehaviour is executed with the respective Workload. Workloads describe the usage intensity of the system (e.g. the event arrival rate or fixed number of users that execute their scenario). A ScenarioBehaviour consists of a sequence of AbstractUserActions. Technically, the latter represents the calls to component interfaces directly accessible by users.

Semantics

At the assembly phase, when the structure of the component instances is defined, the SEFF calls are synthesized into an MSC diagram for each task. The tasks are identified from the UsageScenarios, where each AbstractUserAction creates a task. The usage of hardware resources for each task is reconstructed from the ResourceDemandingSEFF constructs of the components involved in the task execution.

The synthesized model is simulated by a scheduler. The results of the simulation are the detailed timelines of task executions on a processor. Performance properties (latencies, missed deadlines, throughput and resource usage) are extracted from these timelines.

The Palladio method provides the following advantages:

1. well-defined framework of the Palladio component-based architecture;
2. comprehensive set of constructs for parameter-dependent modeling of a component behaviour;
3. explicit modeling of a workload on a system by means of scenario models;
4. detailed behaviour modeling allowing common programming constructions, like conditional statements and loops;
5. usage of distribution functions for accurate performance predictions.

The Palladio method has the following limitations:

1. no modeling support for heterogeneous multi-processor platforms;
2. no support for modeling and analysis of communication latencies and network load;
3. modeling of asynchronous communication style is not supported.

3.3.4 EJB-Liu-Gorton Methodology

The methodology proposed by Liu-Gorton in [72] enables design-time performance prediction of Enterprise Java Beans (EJB) systems built in the client-server architectural style. In this methodology, the EJB components are specified by composable parameterized queueing network models. At assembly time, the component models are synthesized and extended with the platform performance model and application usage profile. The obtained system-wide

model serves as an input to a queueing network evaluation algorithm, which results in predicted load and throughput values.

Syntax

The component behaviour and resource usage is specified in the traditional queueing network terms. Both *Request queue* and *DataSource queue* characterize the communication capacity of a component, while the *Container queue* models the computational capacity of a component. For each queue the following data is specified: the average request population, the size of the available thread pool, and the amount of time required for a request to be served at a queue.

Semantics

The three above-mentioned types of models are combined into the Quantitative Performance Model, which can be solved by traditional QN analysis techniques. The outcome of the analysis includes the predictions on the following properties: the maximum load of the system; average response time with throughput and resource utilization; performance bottlenecks in the system.

The beneficial points of the methodology are:

1. the methodology complies with the COTS standard;
2. explicit modeling of a workload on a system by means of the application profile model;
3. modeling support for distributed systems;
4. modeling and analysis support for communication latencies and network load.

There are a number of limitations imposed by the methodology:

1. supports only client-server architectural style;
2. modeling of synchronous communication style is not supported;
3. component specification is on a very high level of abstraction, therefore the conditional statements and loops cannot be modeled;
4. the QN-based analysis does not provide predictions on detailed behaviour and worst-case latencies.

3.3.5 MPA/RTC

The Modular Performance Analysis/Real-Time Calculus (MPA/RTC) [102] method has been initially developed for component-based streaming applications deployed on distributed systems. The method uses analytic performance evaluation functions and focuses primarily on finding guaranteed worst-case

latencies.

Syntax

The main assumption of the method is that a component executes only one task, receives one input stream of events and sends out one output stream. A component may represent both processing and communication functionality. The function of a component task is characterized by the number of instructions/bytes needed by the function from a processor/network. The method models the hardware resources (processing nodes and communication lines) in terms of their capacity and scheduling policies. At the assembly modeling time, an input stream is specified by an *arrival curve* defining a number of events arriving on an event stream within a specific time interval.

Semantics

The synthesis of the component models of a specific assembly leads to an abstract system performance model. The model is synthesized from the models of individual component models, assembly/deployment model and models of hardware resources. The RTC equations are applied to the performance model, resulting in the predicted values for guaranteed worst- and best-case computation and communication latencies; worst- and best-case usage of processing nodes and communication lines. The RTC concept is to compute sequentially the arrival curves and service curves for each of the components in line that execute the input stream. If multiple components are assigned to one hardware resource, scheduling techniques (TDMA, RMA, DMA) are used to solve the concurrency issues.

The MPA/RTC method provide the following benefits:

1. prediction of guaranteed worst-case latencies and hardware resource load;
2. very low computation complexity;
3. explicit and flexible modeling of a workload on a system by means of event arrival curves;
4. support for modeling and analysis of distributed systems;
5. modeling and analysis for communication latencies and network load;
6. powerful software tools for automation of design and analysis.

The limitations of MPA/RTC are the following:

1. restricted view on a component, as a conventional COTS component may have multiple input/output ports and executes a number of tasks in parallel;
2. modeling of the synchronous communication style is not supported;
3. the RTC equation set does not provide predictions on detailed behaviour, performance bottlenecks and distribution of latencies;

4. very high level of abstraction of component specification does not allow modeling of programming structures.

3.3.6 Alternative Methods for Compositional Perf. Analysis

A number of compositional performance analysis methods has been proposed for particular component-based architectures. The **SaveCCM** component model [46] for real-time embedded systems provides simple modeling and simulation-based analysis facilities. The **KOALA** component-based architecture [100] includes compositional facilities to predict memory usage by a simple analytical technique. The **PECOS** (PErvasive COmponent Systems) component model [34] focuses on extra-functional properties such as memory usage and latencies. The extra-functional properties like memory consumption and worst-case execution times are specified per component basis. The assembly composition and performance analysis are done by different PECOS tools, such as the composition rule checker, the schedule generator and verification tool. The schedule generator simulates the assembly model, providing the predictions on the above-mentioned performance properties. The **Rubus** Component Model [92] also targets resource-constrained systems with real-time requirements. The timing properties of assemblies are analyzed by analytical (schedulability analysis) and simulation-based techniques.

The SaveCCM, PECOS, Rubus and KOALA target systems that employ the pipes-and-filters architectural style. The advantage of these methods is that they use well-specified component frameworks, which can be easily applied to industrial systems. However, they also feature common limitations. The methods provide facilities for modeling of a component behaviour and resource usage at a high abstraction level. Moreover, their performance analysis techniques are relatively simple and restricted to specific application domains.

A number of compositional performance analysis approaches is left out from this survey, but are mentioned here as their contribute to the problem domain. The SOFA [87], HIT-based [104] and REO [3] approaches provide compositional performance analysis based on formal methods, while the NFMDA-based [25], SymTA [48], and CBML-based [103] approaches use semi-formal modeling.

The predictable assembly concept is also widely used for analysis of other system quality attributes. The problem of *reliability* predictions is addressed in the work by Cortellessa [25], Reussner [88] and Triverdi [37, 89]. The compositional analysis methods for *security* properties are proposed by Khan [63], Nicol [78] and Hussein [52]. Grunske [44] proposed a method for specification and evaluation of *safety* properties by using the compositional modeling and analysis.

3.4 Comparison of Predictable Assembly Methods

This subsection presents our comparison of the performance methods described in the subsection above. Besides this, it summarizes the main characteristics of the DeepCompass framework presented in this thesis. Table 3.1 provides the summary of our analysis, where each column refers to a method type and each row specifies an evaluation criterion.

With respect to *Addressed Quality Attributes*, the PECT, KLAPER and SaveCCM provide evaluation of multiple properties, while the rest of the methods focuses on performance. It is important to note that the detailed predictions on the distribution of latencies and resource usage are provided only by the simulation-based methods. At the same time, the worst-case latencies are only provided by the methods based on analytical tools.

The column *Addressed Architectural Styles* specifies the constraints that the methods impose on the architectural style of a system. The PECT, KLAPER, Palladio and DeepCompass methods are generic in the sense that they can be applied to the systems designed in all common architectural styles, e.g. client-server, blackboard and pipes-and-filters. A large group of methods supports only the pipes-and-filters style, which reflects that this style is most commonly used for real-time embedded systems.

None of the methods features a high level of *Modeling Granularity and Effort*. In our vision, component properties, specified at a low level of abstraction[1], duplicate the source code to a certain extend, require a lot of modeling/computation efforts and hinder the securing of IP (Intellectual Property) of a component. Nevertheless, modeling at a low level of abstraction would provide high accuracy in predictions, because then the model would mirror the source code. The methods presented in this section were developed to be applied for complex industrial systems, when the development time and production cost are important factors. Therefore, all the methods given in Table 3.1 solve the efforts-vs.-accuracy trade-off in favor of the low modeling effort.

Regarding the *QA Analysis Type*, the methods use the whole spectrum of available performance analysis techniques. The PECT, KLAPER and Rubus methods combine several techniques to obtain trustworthy predictions.

All the reviewed methods, except for KLAPER and MPA/RTC, are *CBSE-compliant*, which means that their component models adhere to the standards of the CBSE discipline.

In the last decade, *multi-processor heterogeneous hardware platforms* gained high popularity in the embedded systems industry. Such platforms provide de-

[1]The levels of modeling abstraction can be classified into three types: high, medium and low. The high-abstraction models usually specify software entities at the atomic level of components, services or interfaces. The medium-abstraction models specify software entities at the level of operations or method invocations. The low-abstraction models may describe all the detailed programming constructions, like variables, conditional statements and loops.

Table 3.1: *Comparison of methods for compositional analysis of system quality attributes based on the attributes of individual components.*

Method / Characteristics	**Addressed QAs**	**Addressed Architectural Styles**	**Modeling Effort**	**Model Granularity**	**QA Analysis Type**	**CBSE compliance**	**Support for Multi-processors**	**Tooling Support**
PECT	Performance (worst-case latencies, resource load), Temporal safety, Liveness	Blackboard, Client-Server, Pipes-and-Filters	Medium	Medium	QN, Simulation, Model checking	High	Low	Extensive
KLAPER	Performance (worst-case latencies, resource load), Reliability	Blackboard, Client-Server, Pipes-and-Filters	Medium	Medium	EQN, Petri Net, DTMP	Medium	Medium	Low
Palladio	Performance (distribution of latencies, throughput, bus and processor loads)	Blackboard, Client-Server, Pipes-and-Filters	Low	Medium	Simulation	High	Medium	Low
EJB-Liu-Gorton	Performance (worst-case resource load, latencies)	Client-Server	Medium	Low	Traditional QN	High	High	Low
MPA/RTC	Performance (worst-case latencies and resource load)	Pipes-and-Filters	Very Low	Low	Analytic	Low	High	Extensive
SaveCCM	Performance (latencies, static memory usage, processor load), Safety, Reliability	Pipes-and-Filters	Low	Low	Simulation-based	High	Medium	Medium
KOALA	Performance (memory usage)	Pipes-and-Filters	Low	Low	Spreadsheet computation	High	Medium	Medium
PECOS	Performance (memory usage and latencies)	Pipes-and-Filters	Low	Low	Simulation-based	High	Medium	High
Rubus	Performance (task latencies)	Pipes-and-Filters	Low	Low	Simulation-based and Schedulability Analysis	High	Medium	Medium
DeepCompass	Performance (distribution of latencies, throughput, bus and processor loads)	Blackboard, Client-Server, Pipes-and-Filters	Low	Medium	Simulation-based	High	High	Extensive

sign flexibility and optimal distribution of the computation and communication loads among the hardware resource, thereby allowing designers to build efficient systems. The facilities for modeling and analysis of systems built on heterogeneous platforms are only provided by the MPA/RTC and EJB-Liu-Gorton methods.

With respect to the *Tooling Support*, the PECT and MPA/RTC methods provide extensive design toolkits. The toolkits include functionality for storage and retrieval of component models, graphical design of system architectures, model specifications and performance computations.

3.5 Conclusions

In this chapter we presented the concept of predictable assembly, which enables design-time property prediction for systems designed from of a set of arbitrary set of compliant components. Here, we have presented the most stable predictable assembly approaches focusing on their working principles, benefits and limitations. The methods discussed in detail are PECT, KLAPER, Palladio, EJB-Liu-Gorton and MPA/RTC.

The motivation for designing a new framework for compositional performance analysis commences with analyzing the limitations of the existing predictable assembly methods.

- There is only one method addressing the use of critical scenarios. Critical scenarios allow focusing only on relevant behavioural aspects of a system, which significantly reduces modeling and computational effort. This approach is indispensable for analyzing complex systems, which normally have very high number of possible scenarios.
- Only two methods support heterogeneous multi-processor platforms. This kind of platform is now widely accepted for embedded system design.

The combination of these two aspects is not covered in the presented methods, although both aspects are essential to design of modern systems. Apart from the above, a further motivation is the framework should be suitable for industrial applications, where various architectural/communication styles are used. Besides this, in an industrial environment, a rapid prototyping and architecture analysis with low modeling effort is crucial for shortening time-to-market. Summarizing, this has led to the design of a new framework called DeepCompass, featuring the following key properties:

- The usage of critical scenarios and component modeling at a medium level of abstraction results in a low modeling effort and a rapid analysis of architectures.

- Modeling and analysis can be performed for systems built in pipes-and-filters, blackboard and client-server architectural styles.
- The new framework supports modeling and analysis of multi-processor architectures, deployed on an arbitrary hardware topology with homo- or heterogeneous processing and memory blocks.
- The framework is able to predict a broad spectrum of performance properties, like computation and communication latencies, task interleaving and blocking, memory, processor and bus usage.

The next chapter explains our proposal: the DeepCompass framework from the development process point of view, while Chapter 5 specifies all the technical details of the framework. After the presentation of the framework, we will elaborate further on benefits and limitations of our method in relation to the material presented in this chapter.

Chapter 4

DeepCompass Analysis Framework

4.1 Requirements and Design Considerations

In this chapter, we present our DeepCompass (Design Exploration and Evaluation of Performance for Component Assemblies) framework. This framework represents a major contribution of the thesis and proposes a process for predicting system-wide performance quality attributes, based on the quality attributes of individual components.

As a starting point, we specify the objectives of the framework as a set of requirements given below.

- The framework shall enable *rapid* and, at the same time, *accurate performance predictions*. By mentioning rapid performance analysis, we mean the possibility for an architect to design an architectural alternative and to check it against performance requirements within a few days. The accuracy of performance predictions is computed as deviation between predicted and actual values of performance properties. These deviations should be within the order of several tenths of percents. We are prepared to accept this level of deviations, because the demand for rapid analysis is more important at the early architecting stages in industry.
- The framework shall require reasonably low modeling and computational efforts.

- The framework shall target a broad selection of performance properties. The selection should include computation and communication latencies, task interleaving and blocking, as well as memory, processor and bus usage.
- The framework shall support modeling and analysis of systems built in all common architectural styles, e.g. pipes-and-filters, blackboard and client-server styles.
- The framework shall support analysis of systems deployed on an arbitrary hardware topology with homo- or heterogeneous processing nodes.
- The method shall be compliant with the CBSE principles and shall be applicable to any component-based software technology, i.e. CORBA, Koala, RUBUS, ROBOCOP or Java Beans.
- The framework shall provide an architect with guidelines for further architecture optimization with respect to its multiple quality attributes.
- The framework shall automate complex computational operations and provide graphical means for designing the alternatives and for visualization of the analysis results.

The following two paragraphs outline the DeepCompass framework. From the point of view of an architectural process, the framework addresses all phases of system design and analysis. It helps an architect to pass through the phases of the architecting process with a focus on performance quality attributes. The DeepCompass framework proposes the following iterative development paradigm. First, rapid building of a number of alternative architectures from available software and hardware components. Second, synthesizing the models of these individual components into a system-wide performance-related model for each alternative. Thirdly, analyzing the performance models of the obtained architectures and encountering the bottlenecks and the points where an architecture does not comply with the requirements. These three high-level architectural phases save effort, help selecting promising architectural alternatives and elicit the correct directions for their fine tuning. Finally, the iteration cycle concludes with refinement of promising architectures to obtain better performance, while keeping other properties (e.g. reliability, cost of materials) within defined boundaries.

From the technical point of view, the cornerstones of the framework are composable models of individual software components and hardware blocks. The models are specified at component-development time and shipped in a component package. At system design, i.e. component-composition phase, the models of the constituent components are synthesized into a system model. Depending on the types of models used at the synthesis, the system model can represent various system-wide properties, e.g. security, reliability or system cost. Since the thesis focuses on performance properties, we introduce performance-related types of component models, such as behaviour, perfor-

mance and resource models. These models, when synthesized together, form an executable system model. The executable model represents the dynamic system architecture and contains detailed data about tasks running in that system. Further simulation-based analysis of the obtained executable system model allows acquiring application-specific and system-specific performance property values. The analysis helps to reason about the quality attributes of a particular architectural alternative. A Pareto-based comparison between the designed alternatives allows selecting the optimal architectures, and serves as a basis for the subsequent design iteration.

In this chapter, we present the framework from the point of view of an architecting process. Fig. 4.1 positions Chapter 4 within the thesis. Besides presenting the framework, this chapter prepares the reader also for the subsequent two chapters that address the technical aspects of the framework, namely the scenario-based simulation method and the architecture optimization approach.

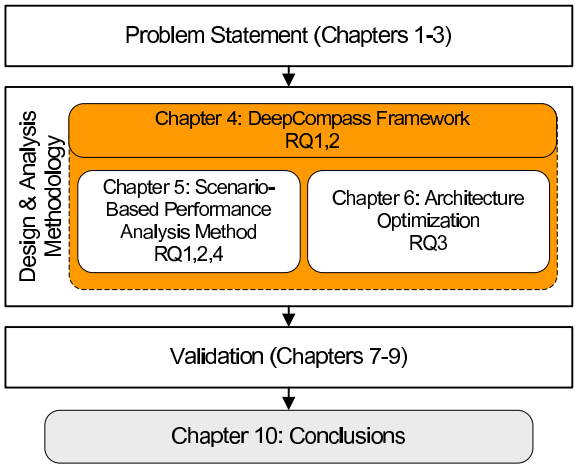

Figure 4.1: *Position of Chapter 4 in the thesis and its relation to Chapter 5 and 6.*

The chapter is structured as follows. Section 4.2 gives an overview of the DeepCompass framework from the development-process point of view. Section 4.3 describes the modeling phase of the process. Section 4.4 presents the architecture and design phase. Section 4.5 provides insights into the analysis and validation phases, while Section 4.6 discusses the comparison of the architectural alternatives and trade-off analysis solutions. Section 4.7 concludes the chapter.

4.2 Overview of DeepCompass framework

Current software development processes include a number of stages, such as requirement analysis, architecture and design, implementation, testing and validation. Fig. 4.2(a) presents the iterative development process [58] that includes these stages conducted iteratively until system requirements are satisfied. In this iterative process, system validation against requirements is performed once the implementation and testing stages are complete. This may increase development efforts in case the validation fails. In order to save on these efforts, we advocate an iterative architecting sub-process, that allows obtaining the correct architecture prior to proceeding to the implementation stage. Fig. 4.2(b) presents this iterative architecting sub-process in the context of the whole development process.

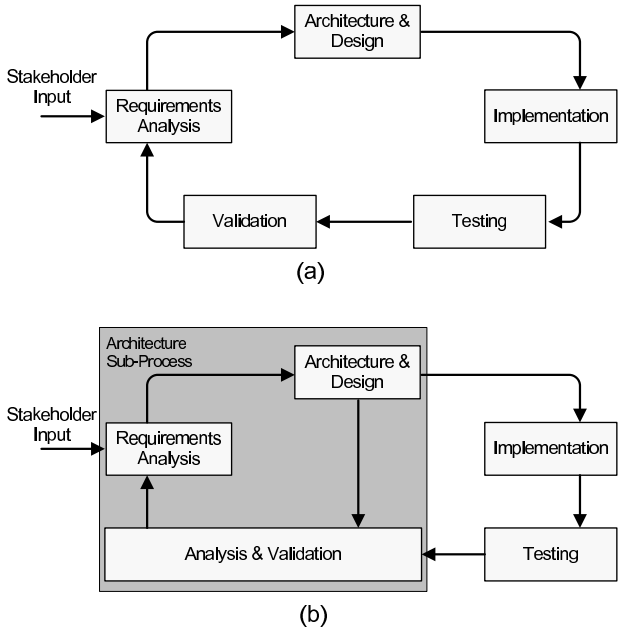

Figure 4.2: *(a) Common iterative process of software development; (b) Iterative architecture process for time-critical systems.*

The architecting sub-process includes three stages of requirements analysis, architecture, as well as analysis and validation. The sub-process iterations may include changes in requirements and architecture, based on the results from the analysis and validation stage. These iterations help an architect to tune the architecture until its predicted properties satisfy the requirements. The implementation and testing stages have to be performed only once after the

architecture satisfies the requirements. This eliminates costly repetitions of implementation and testing.

Fig. 4.3 portrays the detailed diagram of the DeepCompass architecting process. The *Modeling* phase targets the specification of models for software component and hardware blocks. The models can be specified at component development time, i.e. prior to the architectural phases. The *Architecture and Design* phase aims at the development of a number of architecture alternatives that would potentially satisfy the *System Requirements*. Besides this, for each alternative, this phase synthesizes the models of individual components into an executable system model. The *Analysis and Validation* phase features prediction of the system behaviour and performance by simulation of the obtained executable system model. The *Trade-Off Analysis for Alternatives* phase aims at comparison of the defined alternatives accounting for the trade-offs between the following multi-dimensional criteria: latency, throughput, robustness and hardware resource load. In the sequel, we explain the above-mentioned phases in more detail.

To support an architect to smoothly pass through these framework phases we have developed a CARAT software toolkit. The toolkit is an integrated set of the following applications:

- repository of components and their models;
- graphical tools for creating software and hardware architectures;
- synthesizer of an executable system model;
- model-simulation engine;
- performance results visualizer.

The description of the CARAT toolkit is given in Chapter 7.

4.3 Modeling Phase and Repository

A. Modeling Phase
The main objective of component modeling is to create a specification that addresses a specific aspect of a component, abstracting from irrelevant details. During our research on defining the model structures, an important requirement was that models of the same type should be composable. We use the ROBOCOP component-based technology as a starting point for the model specification, as well as for the validation of the performance analysis framework. The framework is not limited to only ROBOCOP components but can be applied to any component-based technology that has a notion of *provided* and *required* interfaces. A formal specification of the introduced models is given in Chapter 5.3.

All the model types we define here, enable parameter-dependent specification [16]. For software components, the framework introduces three types of

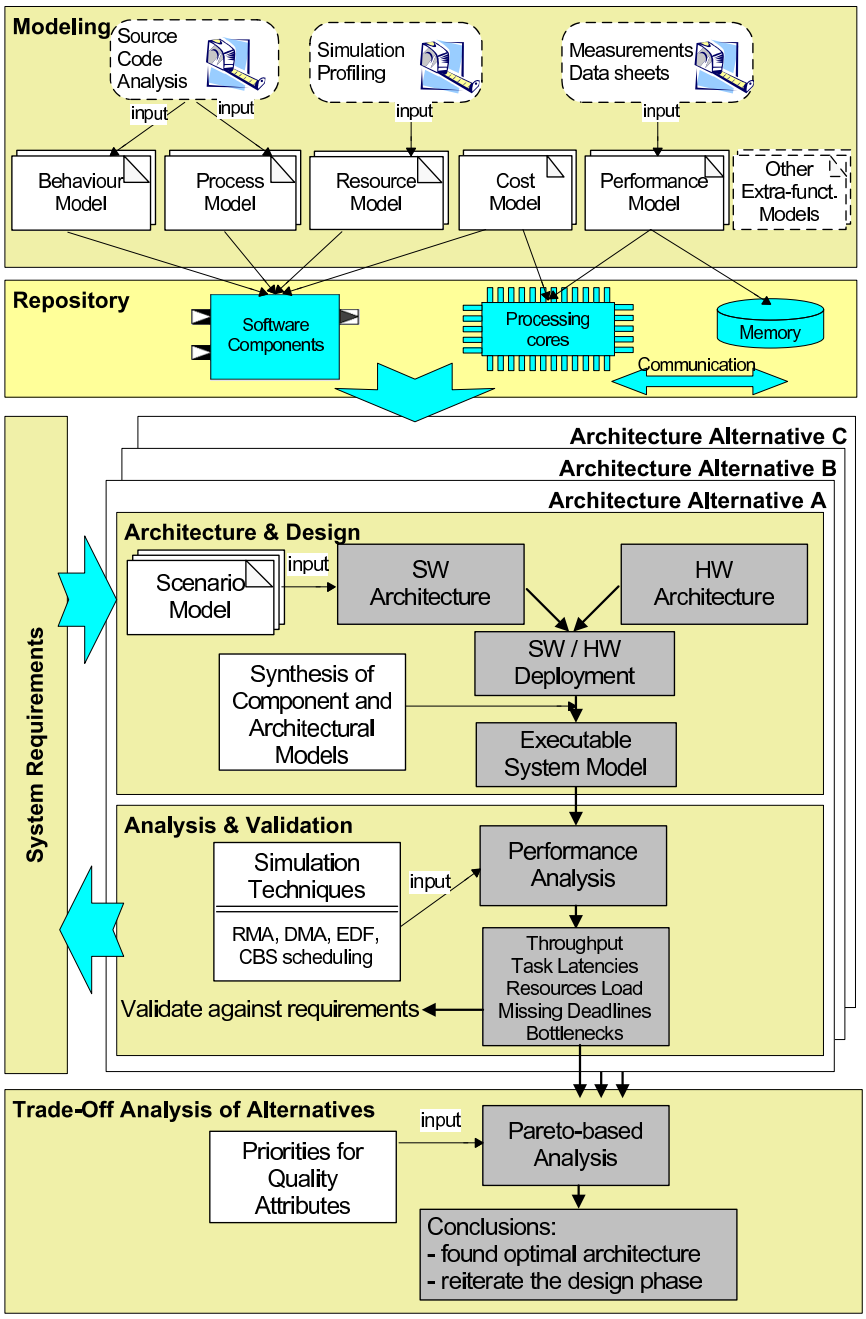

Figure 4.3: *Detailed view on the DeepCompass architecting process.*

models: resource, behaviour and process models. Typical models for hardware IP blocks (hardware parts) are performance models for memory, networks and processors. Our experience showed that these models can be made with comparatively little effort (discussed in Chapter 9) to achieve reasonable prediction accuracy.

The *resource model* specifies parameter-dependent resource requirements (e.g. number of claimed processing cycles) of each individual operation of a component. The resource requirements can either be obtained by estimations or by profiling of each individual component on a reference processor. The characteristics of the reference processor should be specified in this model, in order to scale the operation resource requirements to any other processors.

The *behaviour model* is used to describe the behaviour of passive components (see the discussion on passive and active components in Section 2.2.2). The main objective of the behaviour model is to specify parameter-dependent behaviour of operations provided by an individual component. The model describes the implementation of a component operation at a high abstraction level. For each operation of an individual component, the behaviour model defines the invocation sequence of operations of other components, that the individual component calls via his required interfaces.

The *process model* is used to describe the behaviour of active components. The process model specifies the processes activated and running within an active component. These processes may activate task instances (like the triggers discussed later in the scenario model), that propagate outside the component boundaries. For every process, this model describes the process creation and release conditions, periodicity, and a sequence of underlying operation calls to other interfaces made by the process. The data for the behaviour and process models can be obtained from design specifications or by a source-code analysis.

The DeepCompass framework introduces *performance models* for hardware resources (processor core, memory and bus). A performance model for a processor core defines its instruction type (RISC, CISC or VLIW) and clock frequency. A performance model for a memory IP block describes the memory type (SRAM, SDRAM, etc.), and the memory capacity in MBytes. A bus-performance model specifies the scheduling protocol (TDMA, CDMA, fare use, etc.) and the bandwidth size. The data for the performance models can be obtained by measurements or from the supplier data sheets.

The *cost model* is a one-line specification of the price for a software component or a hardware node. The price can refer to the purchasing cost, the development/material costs or a mixture of those.

The ROBOCOP architecture is open with respect to the model types for analysis of extra-functional properties. Therefore, in the DeepCompass framework we assume that *other extra-functional models* (reliability, security, safety, etc.) can be added later to further extend the analysis (see Fig. 4.3).

The requirements for defining a new model type are as follows: (a) the

new component model should enable compositionality such that the synthesis algorithm (specified in Chapter 5) should be able to compose the models of individual components into a system-wide model, (b) the modeling level of abstraction should be high enough to keep the modeling effort at a reasonable level, e.g. a model should not repeat a source code, but only describe its important/relevant characteristics.

The following important constraints are applied to a new model type. First, the model structure should be specified in the EBNF notation provided in Section 5.3. Second, new modeling primitives should be associated with operations provided by a component. These associations will be used for the synthesis algorithm to construct a system-wide model. Finally, a new model should use the same atomic datatypes (string, boolean, integer, etc.), or should provide explicit definitions for a new datatypes.

B. Repository
The component models need to be specified at component development time, committed to a component package and stored into a common *repository*. The repository stores component packages from third-party developers and provides access to multiple system-development parties. The repository is not a phase of its own, as it only provides the infrastructure for the components and their models.

4.4 Architecture and Design Phase

The goal of the architecture and design phase is to define *a set* of software and hardware (SW/HW) architectures that would satisfy the system's functional and performance requirements. An architect may define a number of such architectural alternatives for future comparison and trade-off analysis. For each alternative, the architecture and design phase should be carried out individually. As input to the phase, an architect has system requirements and various third-party hardware and software components stored with their corresponding models in a repository. The following subsections describe the actions of the design phase, starting from the specification of scenario models and ending with the synthesis of an executable executable model.

4.4.1 Scenario Models and Software Architecture

An architect selects from a repository software components that, when composed together, *should* satisfy functional requirements and *may* satisfy extra-functional, e.g. performance requirements. The component selection is done by checking the functional models of available components with respect to the

functional requirements. We assume that the selected components are supplied together with their corresponding set of models.

By means of the CARAT graphical tool, an architect graphically defines a component composition by instantiating and connecting the selected software components. This component composition, together with data on external task-triggering events, is stored as a *scenario model*. Hence, the scenario combines the specification of a component composition and its input events (task triggers or stimuli). These input events enter a system and cause some activity within a system. The input events can be of the following types: user-based (push-button action), environmental (signal to a sensor), platform interrupts and messages from other systems. For an event, an architect may define the period, deadline, offset, jitter, corresponding task priority, and so on. The specification of such input events allows an architect to characterize a set of scenarios describing the ways a system should be used.

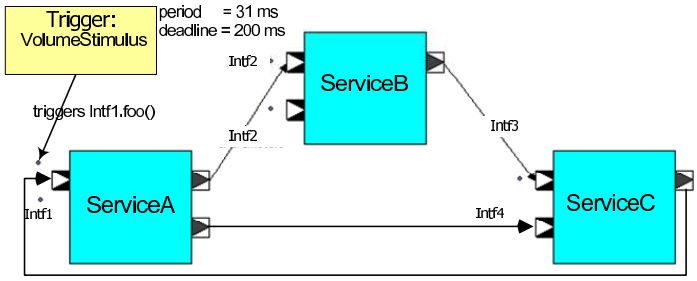

Figure 4.4: *Simple example of scenario model.*

Fig. 4.4 depicts an example of a scenario model. The component composition in this model consists of three individual components (`ServiceA`, `ServiceB` and `ServiceC`) bound via interfaces `Intf1`, `Intf2`, `Intf3` and `Intf4`. The model also contains one trigger `VolumeStimulus`, which triggers the operation `foo` provided by the interface `Intf1`. An architect may also specify the deadline and periodicity parameters for this trigger.

Often, the scenarios are also called 'usage profiles' or 'use-cases'. The scenarios are introduced in order to avoid modeling of the complete set of possible system configurations and environmental events, and to concentrate on the critical execution scenarios that may pose resource overload or missed deadlines. A specific type of scenarios commonly found in literature and in practice are worst-case scenarios. Guidelines for identification and selection of the critical scenarios are presented in Chapter 8. An individual scenario is described by a separate scenario model. Each individual scenario model is a subject for further performance analysis. The detailed structure of a scenario model is specified in Section 5.4.2.

4.4.2 Hardware Architecture and Deployment

A hardware-architecture specification should be developed as part of the framework. Often in a project, a hardware platform is pre-determined. In this case, an architect needs to specify the hardware platform in terms of the DeepCompass hardware architecture model. If an architect has freedom to define a hardware architecture, he can select hardware blocks from a repository and define a desired topology. The topology specifies processing nodes, memory blocks, and their communication means, i.e. bus lines. The selection criteria are left to the designer's intuition. The topology can be graphically designed using the CARAT editor (see Section 7.2.2).

Once the software and hardware architecture are defined, a deployment of the software components on the hardware nodes can be made. The SW/HW deployment describes the assignment of each software component on individual processing nodes. In other words, the SW/HW deployment model shows for each processing node, which software components are executed.

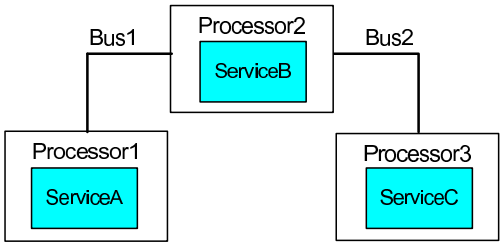

Figure 4.5: *Simple example of hardware architecture and deployment model.*

Fig. 4.5 presents a simple example of a deployment model. Here, the hardware topology contains three processing nodes Processor1, Processor2, and Processor3, which are connected via communication lines Bus1 and Bus2. The SW/HW deployment model defines that the software component ServiceA is executed on Processor1, ServiceB is executed on Processor2, and so on.

Efficient deployment is required to distribute the load of hardware resources in an optimal way. However, at the first deployment iteration, it is not clear how to deploy the software components to achieve the optimal load distribution. Therefore, various deployment alternatives are possible and each alternative represents a system architecture that can be assessed against the requirements at later phases. The deployment can be defined using the CARAT graphical editor. The CARAT toolkit converts the deployment diagram into the deployment model. The detailed structure of the deployment model is given in Section 5.4.3.

4.4.3 Executable System Model

At this point, an architect obtains a number of various models, such as the deployment and scenario models, as well as the models of individual software components and hardware blocks. So the task-related data, required for performance analysis, is spread over different types of models. For instance, the *task periodicity* is specified in the scenario model, whereas the information about the *operation call sequence* comprising the task is spread over corresponding component behaviour models. Therefore, the models need to be synthesized in order to obtain an analyzable task-execution architecture of a system.

The DeepCompass framework defines a set of model composition rules, that enable obtaining a system-wide execution architecture. These composition rules, implemented by the CARAT toolkit, check for completeness and consistency of all the input models and synthesize their corresponding models into an *executable system model*. The executable model describes a set of tasks (running in a system) with a detailed description of: (a) a sequence of operations and interactions executed by each task, (b) execution time and communication load imposed by each operation within a task, (c) synchronization constraints between the tasks, and (d) task period, jitter, offset and deadline.

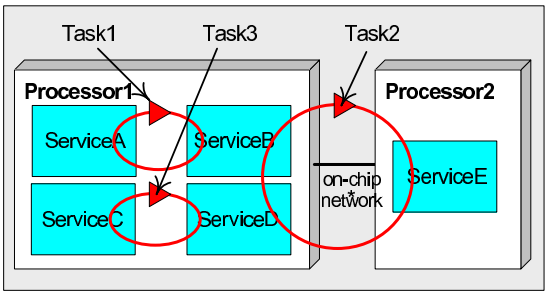

Figure 4.6: *Example of task mapping on the SW/HW architecture blocks.*

The executable system model also allocates these tasks to the software components and hardware blocks. An example of this allocation is given in Fig. 4.6. Here, the system executes three tasks, using two processors and five deployed service instances. Task1 executes on Processor1 and consists of operations offered by ServiceA and ServiceB. The execution of Task2 is spread over both processors and includes a communication via the on-chip network. This task executes operations offered by three service instances: ServiceB, ServiceD and ServiceE. The data transfer between the service instances B, D and E is mapped onto the on-chip network.

Detailed description of the model composition rules and executable system model is given in Sections 5.4.4 and 5.5.

4.5 Analysis and Validation Phase

The synthesized executable system model is used for obtaining the performance properties. A number of performance analysis approaches can be applied to this model. The approaches can be categorized in two types: schedulability analysis and simulation-based techniques. The schedulability analysis results in predicted worst- and best-case response times of each task and overall CPU, memory and bus utilization for a specified scenario. The simulation-based techniques yield richer output, including task-execution behaviour (patterns of execution), distribution of latencies for each task, and fine-grained (in time) resource utilization for a specified scenario.

The DeepCompass framework enables both types of analysis. However, in this thesis we focus on simulation-based techniques, because they output extensive data on the task-execution behaviour and latency distribution. This data can be used for thorough performance analysis, e.g. diagnostics of bottlenecks. DeepCompass provides (by means of the CARAT toolkit) a set of schedulers that simulate parallel execution of the tasks specified in the executable system model [71]. The selection of scheduling algorithms is dictated by the types of networks and operating systems used for the designed system. The CARAT toolkit provides the following schedulers: rate-monotonic (RM) [8], deadline-monotonic (DM), earliest deadline first (EDF) [23], time-division multiple access (TDMA) and fair-use algorithms. The simulation techniques feature scheduling of both processing and communication resources.

The simulation of the executable model results in predicted performance properties for each of the architecture alternatives. These properties may include throughput, task latencies, number of missed deadlines, utilization of hardware blocks, and need to be validated against the system performance requirements. Should the properties fail to meet the requirements, then an architect can re-iterate the architecture and analysis phases, or skip this architecture alternative. During the next iteration, an architect may: (a) evaluate other available software components and hardware IP blocks, (b) make different SW/HW deployment, (c) apply other scheduling policies, etc.

The throughput, latency and resource-consumption properties can be extracted in a straightforward way from the task execution-timeline obtained by simulation. For other attributes, like *robustness* and *reliability*, additional computations are needed. Robustness can be calculated as the performance sensitivity to an increase of the input event frequency (caused by a trigger). To obtain this sensitivity, an architect reduces the trigger periods in each of the scenario models and repeats the simulations. Comparison of the new task latencies or resource use with the old values shows how sensitive the architecture is against the changes of the input event rate.

In Section 5.6, we discuss the simulation techniques and performance analysis in detail.

4.6 Trade-Off Analysis for Alternatives

Once a number of alternative architectures has been defined and their performance properties are obtained, an architect looks for an optimal architecture alternative. Finding an optimal alternative with respect to multiple properties (objectives) is a challenging task. The objectives often conflict with each other, e.g. in order to achieve higher performance, an architect needs to use more expensive hardware. A trade-off analysis helps to understand and solve these conflicts. The DeepCompass framework allows the designer to use a Pareto analysis as a powerful means for resolving conflicting objectives [74].

Generally, Pareto-based optimization methods do not yield a unique solution, but a set of solutions that are Pareto-optimal. The selection of the preferred alternative from this set is based on the quality-attribute priorities defined by an architect. For instance, in Fig. 4.7, we depict a Pareto diagram for finding optimal architectures with two-dimensional criteria: *system cost* and *critical task latency*. Each architecture solution is placed in the diagram according to its attribute values.

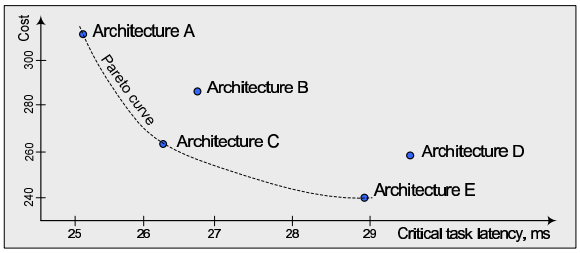

Figure 4.7: *Two-dimensional trade-off analysis diagram with various architecture alternatives and indicating the Pareto curve.*

A Pareto curve is depicted by connecting the alternatives that are closest to the origin. This curve defines a set of optimal alternatives. The Pareto curve in Fig. 4.7 passes through the optimal alternatives A, C and E. The other solutions B and D (above/to the right from the curve) are not optimal. Obviously, Architecture B is worse than Architecture C, because it is more expensive and slower. The same holds for Architecture D. Further analysis can be done for the optimal architectures. If, for instance, the timing requirement for the critical task is 29 ms, then Architecture E can be rejected as a risky solution. If the system cost is a dominant attribute, than we may consider Architecture C and E as preferred optimal solutions. Evaluation of three- and four-dimensional trade-offs is more challenging for visualization, but conceptually the same principle holds.

The selected optimal architecture alternative that satisfies system require-

ments can be either sent further to the implementation stage (see Fig. 4.1(b)), or re-iterated, in order to achieve an even better combination of quality attribute values. The detailed discussion on the trade-off analysis and architecture optimization is given in Chapter 6.

4.7 Conclusions

This chapter started with a discussion on the common processes for development of software systems. We have presented an overview on a common iterative software development process and encountered its limitation for the development of real-time systems. The limitation is that the architecture and validation stages are separated in time by the implementation and testing stages. As a consequence, for each iteration, suboptimal or even incorrect design decisions in the architecture may be identified too late, only after implementation and testing. This latency may cause project delays and budget overruns. We have proposed to extend the iteration loop with a sub-process that comes to performance properties, while performing only a requirements analysis, architecture design and analysis/validation stages. The sub-process enables prediction of the architecture performance properties and their validation against the requirements without the need for implementation and testing.

The DeepCompass framework guides an architect through this sub-process. The sub-process includes the following phases: (a) creation of the software and hardware architectures and generation of the corresponding design-model specifications, (b) synthesis of all the design-models and models of individual components into an executable system model, (c) analysis of the executable model and analysis of the obtained performance properties, and (d) multi-objective trade-off analysis and comparison of the architecture with other alternatives.

Summarizing, the DeepCompass framework has the following key features.

- The analysis of a system composed from third-party components can be done without buying these components. The analysis is based on the component models, which may be distributed free of charge.
- The synthesis of the system-wide model, representing tasks running in a system, is completely automated. We constructed the structures of different component models to be composable with each other resulting in a system-wide model.
- The framework features *rapid* architecture prototyping and performance assessment. This is achieved by provided graphical design-tools; automated model synthesizer and an efficient simulation engine.
- The framework requires low modeling and computational efforts. This is achieved by accurate selection of modeling primitives and a relatively high abstraction level of individual component models. The usage of scenarios for software architecture modeling also reduces efforts, because

an architect focuses only on performance-critical scenarios and leaves the scenarios that do not influence performance properties aside.
- The framework addresses a wide spectrum of performance properties, such as latencies, task interleaving and blocking, as well as memory, processor and bus usage.
- The framework supports modeling and analysis of systems built in all common architectural styles, e.g. pipes-and-filters, blackboard and client-server styles.
- The framework supports analysis of systems deployed on an arbitrary hardware topology with homo- or heterogeneous processing nodes.

The following chapter describes how these benefits are achieved. In that chapter, we present the detailed specifications of (a) structures of individual component models, (b) algorithms for synthesis of different model types into an executable system model, and (c) the simulation engine for processing this system model and obtaining the performance properties of the system. When all of these details are discussed, we also address the limitations of the framework.

Chapter 5

Scenario-Based Performance Analysis Method

5.1 Introduction

In the previous chapter, we have presented the DeepCompass framework that defines an iterative process for design and validation of performance properties of architectural alternatives. The core of the framework is the compositional scenario-based method for performance analysis. This chapter presents this method in more detail, builds further on the DeepCompass framework and concentrates on the research questions posed in Chapter 1 of this thesis.

RQ1: How should behaviour and performance properties of individual components be specified in order to enable composition of these properties into an analyzable model of a complete system?

RQ2: How to combine the models of individual components into the model of a complete system in an automated way, such that the resulting system model can be analyzed with respect to the performance properties?

Besides these general research questions related to an automated synthesis of assembly models, this chapter concentrates also on the following detailed questions.

- What kind of component-level models and architectural-level models should be introduced in order to cover all performance-relevant aspects of a system? The key constraint is that these models should be concise and easy to read and construct.

- Which performance and behaviour properties of software components should be specified in a model (and which can be safely omitted) in order to obtain accurate predictions of performance properties of a system?
- How to enable modeling of systems with parameter-dependent behaviour and resource usage?
- What modeling primitives are needed to allow a component developer to model passive and active components? Another issue is which primitives enable an architect to successfully model systems designed using different architectural styles (pipes-and-filters, blackboard and client-server)?
- How to model the component usage of processors and busses in a hardware-independent way? For example, how to specify already at the component development phase, the time needed for a component execution on an arbitrary processor?
- How to enable modeling of an arbitrary hardware platform with heterogeneous processing nodes? This question deals with specification, while the next one is about analysis of this specification.
- How to simulate the execution of an arbitrary set of software components mapped on an arbitrary (incl. heterogeneous) hardware topology?
- How to allow an architect to specify a deployment scheme, defining which software component is executing on each processing node?
- How to enable analysis of only those system execution-configurations that are relevant to performance requirements?

Contributions of our method are definitions of both the *composable structures of performance-related component models* and the *algorithms for the automated model synthesis*. Another important contribution of the method is the introduction of the *scenario model*. The scenario model is a combination of a component composition (i.e. static software architecture) together with a characterization of possible input events to the system. These events, being analyzed and modeled by an architect, define a potential workload on a system.

The outcome of the synthesis of the above-mentioned models is an executable system model that describes system-wide behaviour. In detail, this model specifies parallel tasks that a system is going to schedule and execute. A *task* is a logical representation of a system activity, similar to an operating system process, which can be started by a system external or internal stimulus (trigger). From the execution point of view, a task consists of a sequence of operations of different components, which can be mapped onto different hardware nodes.

Finally, the method defines a number of algorithms to simulate the obtained executable system model. Such a simulation produces task-based timelines, from which the following performance properties can be obtained: task latencies, throughput, hardware resource loads and even robustness of the system against system overload conditions.

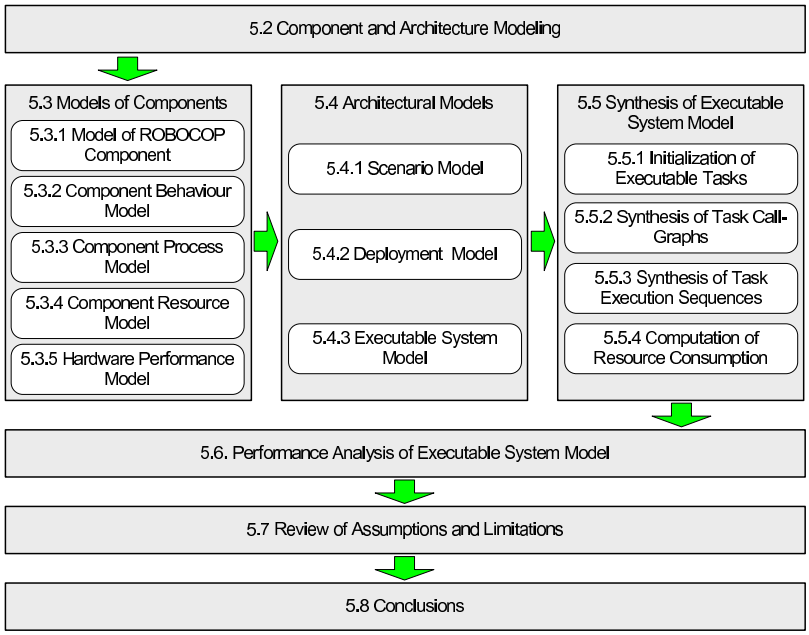

Figure 5.1: *Structure of Chapter 5.*

Fig. 5.1 portrays the structure of this chapter. Section 5.2 introduces the DeepCompass model types for individual components and for the architecture specification. Besides this, we compare and map these model types onto the model types from the common 4+1 Architectural View paradigm. Section 5.3 specifies the individual component models in the EBNF grammar notation. These include the following: ROBOCOP model, component behaviour-, process-, resource models and hardware performance model. In Section 5.4, we present the models for architecture specification such as scenario-, deployment- and executable system models. Section 5.5 explains the algorithms for automated model synthesis resulting in the executable system model and provides examples for each synthesis step. Performance analysis by simulation of the latter model is presented in Section 5.6. Section 5.7 outlines the assumptions and limitations of the presented models and performance analysis method. Section 5.8 concludes the chapter.

5.2 Component and Architecture Modeling

Within the DeepCompass framework, modeling activities are carried out in two phases: (a) at component development phase and (b) at system design phase.

The models specified in the component development phase aim at describing particular characteristics of individual software components and hardware IP blocks. The objective of the models created at the system design phase is to specify a system architecture, in terms of the relations between software components and their deployment onto a hardware architecture.

For a software component, we introduce the following three model types: resource-, behaviour- and process models. Each of these model types aims at an abstract specification of corresponding component properties. These models can be used for two purposes: (a) to specify requirements on a component to be developed, and (b) to reflect properties of an already implemented component.

The general characteristics of individual hardware IP blocks such as processing nodes, memory modules and communication means are described in a HW performance model. This model includes hardware parameters, e.g. processing power and instruction type, memory capacity, or bus bandwidth.

The models that describe the outcome of a system design phase include a software scenario model and a deployment model. These models specify both software and hardware architectures, as well as the mapping of software components onto the hardware nodes.

The set of above-mentioned models has been defined to capture all necessary data needed for performance analysis. These models aim at similar objectives as the widely used 4+1 View Model [85]. This model describes a software architecture using five complementary views, each of which addresses specific concerns. The *logical view* describes the system structure in terms of classes or components; the *process view* describes the system concurrency and synchronization aspects; the *physical view* describes the mapping of the software onto the hardware and shows the system's distributed aspects, and the *development view* describes the software-code organization in the development environment. An architect can organize the description of the architectural decisions around these four views and then illustrate them with a few selected *use cases*, thereby constituting a fifth view.

Motivation for another set of models

The 4+1 View Model is valuable for design of general-purpose software systems. However, it does not provide specific support for the domains of component-based software architectures and time-critical systems. It does not allow specification of components and system-wide performance properties. Another limitation is that the 4+1 View Model focuses only at the specification of a complete system, while we aim at both the individual components and system-level specifications. Finally, the 4+1 model types do not provide performance-oriented design facilities in an integrated way.

To support convenient reading, we have mapped the DeepCompass model types onto the types from the 4+1 View Model. Fig. 5.2 depicts conceptual similarities between the DeepCompass models and 4+1 View Model. Our *sce-*

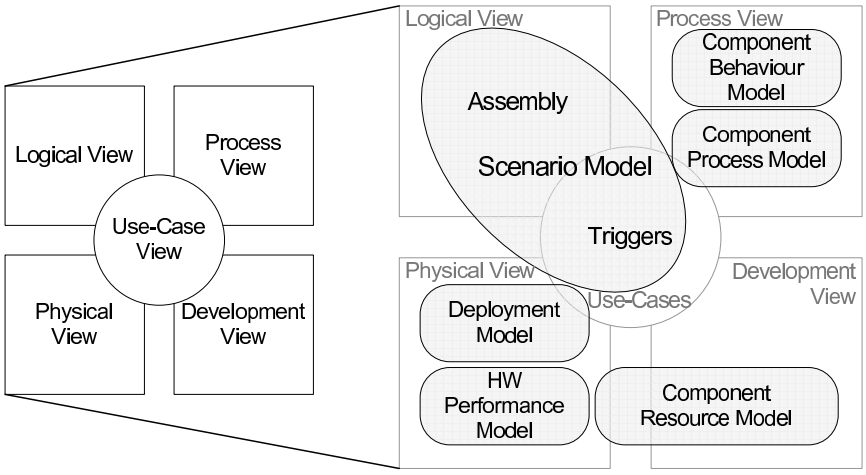

Figure 5.2: *Mapping of DeepCompass models onto the 4+1 architectural view paradigm.*

nario model addresses similar issues as the logical and use-case views. The component assembly specification and the logical view represent a static architecture organization. The triggers, described in the scenario model, are conceptually similar to the use cases representing system workload configurations. The *deployment model* is similar to the physical view due to the fact that they both specify (a) the hardware nodes and architecture topology and (b) the mapping of the software entities on the hardware nodes.

Our *component behaviour/process model* and the process view both specify the behaviour of a software entity. The difference is that the behaviour/process model describes the behaviour of individual components, while the process view shows the system-wide behaviour. Similarly, the *HW performance model* specifies characteristics of individual hardware IP blocks, while the physical view may describe system-wide hardware characteristics. Another difference is that the 4+1 Model View does not provide similar means as the DeepCompass *resource model*. These differences lead to the conclusion that the set of DeepCompass models is more comprehensive for performance analysis, and features an executable system model. The 4+1 View Model lacks these facilities.

5.3 Models of Components

This section presents a formal specification of the four component model types and shows one graphical example for each type. The formal model definitions are constructed by means of the Extended Backus Naur Form (EBNF) [33].

5.3.1 Model of ROBOCOP Component

In Section 2.2.3 we have outlined the definition of the ROBOCOP architecture and its component model. In our approach, a component incorporates an open set of models including an executable model, a source-code model and a functional model. The executable model specifies the (a) services of the component (that can be instantiated), (b) interfaces that provide access to the functionality of the services and (c) operations that implement this functionality and which can be invoked by other services through the respective interfaces.

Model of Robocop Component

robocop-component	= component-id, component-name, model-set
model-set	= executable-model*, behaviour-model, process-model, resource-model
executable-model	= service*
service	= service-id, service-name, provided-interface*, required-interface*, is-buffer
provided-interface	= interface-type, pp-name
required-interface	= interface-type, rp-name
interface-type	= interface-name, interface-id, operation*
operation	= operation-name, operation-id, passed-arg*, returned-arg*
passed-arg	= arg-name, arg-type
returned-arg	= arg-name, arg-type
component-id	= GUID
service-id	= GUID
interface-id	= GUID
operation-id	= GUID
component-name	= CHAR*
service-name	= CHAR*
interface-name	= CHAR*
operation-name	= CHAR*
arg-name	= CHAR*
pp-name	= CHAR*
rp-name	= CHAR*
arg-type	= “boolean” | “byte” | “short” | “integer” | “long” | “double” | “float”
is-buffer	= BOOLEAN
GUID	= {“0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9” | “A” | “B” | “C” | “D” | “E” | “F”}[32]

Figure 5.3: *EBNF definition of the model of a ROBOCOP component with its executable model included.*

The EBNF definition of the model of a ROBOCOP component is given in Fig. 5.3. As it is the first model described in the EBNF syntax, we give an extensive interpretation of it below.

The `robocop-component` consists of the component's `component-id`, `component-name` properties and includes a `model-set`. The `model-set` consists of zero or more `behaviour-model`, `process-model`, `resource-model` and zero or more `executable-models`. The model set is not limited to these models.

The specification of the former three models is given in the next subsections.

The `executable-model` specifies a set of `services` that provide actual functionality. A service is defined by a `service-name` and a `service-id`. An `is-buffer` boolean element defines if the service is a buffer entity that represents a memory space, or, otherwise, if it is an executable entity that contains executable code. The buffer-type services are purely used for the pipes-and-filters architectural style and are actually pipes in that context.

A service provides its functionality via a set of `provided-interfaces`, that other services can connect to. A service also defines a set of `required-interfaces`. These are the interfaces to which a service needs to be bound in order to operate properly.

Each `provided-interface` contains a description of the specific `interface-type` that it implements. Because a service may provide a number of `provided-interfaces` of the same `interface-type`, these are distinguished by a provided port name (`pp-name`) element. This is a service-specific name for each port. A port can be seen as an instance of an interface. The `required-interface` specifies the `interface-type` of the provided interface to which it should be bound.

The `interface-type` itself is defined by `interface-name` and `interface-id`. Besides this, it defines a set of zero or more `operations` (i.e. methods, functions), that this service should implement.

An `operation` is defined by an `operation-name` and `operation-type` pair. Furthermore, an `operation` specifies a set of passed and returned operation arguments - `passed-arg` and `returned-arg`. An argument is defined by an `arg-name` and `arg-type` pair.

The identifiers `component-id`, `service-id`, `interface-id` and `operation-id` are specified as a `GUID`, which is a Globally Unique IDentifier consisting of 32 hexadecimal numbers. The rest of the model defines so-called modeling primitives. For instance, the `component-name` is a sequence of the `CHAR` type. The `arg-type` can be one of the following: `"boolean"`, `"byte"`, `"short"`, `"integer"`, `"long"`, `"double"` or `"float"`.

The component models, introduced further in this chapter, use the definitions from Fig. 5.3. The concepts of the ROBOCOP component model are compatible with common component models.

5.3.2 Component Behaviour Model

This section first introduces the EBNF-based model description and then presents an example of what the model actually describes. The main objective of the behaviour model is to specify parameter-dependent behaviour of operations provided by an individual passive component. The model should describe implementation of a component operation at higher abstraction level. For each component operation the model defines the invocation sequence of

operations of other components, that the component calls via his required interface. For more detailed modeling of loops and conditions in the source code, the model allows to specify iterations (`for`, `while-do` loops) and conditional statements (`if-else`, `switch`). Besides that, the model specifies the data transfer activities from one service to another. In source code, these data transfers are commonly programmed as passed and returned arguments in the invoked operation signatures.

Component Behaviour Model

behaviour-model	= operation-behaviour*, parameter*
parameter	= parameter-name, parameter-type, parameter-id, instantiation-constraints
instantiation-constraint	= min-possible-value, max-possible-value, [default-value]
operation-behaviour	= operation-id, behaviour-description, mutex-mode
behaviour-description	= {parameter-range*, call-graph}*
parameter-range	= parameter-id, min-value, max-value
call-graph	= execution-loop*, critical-section*
execution-loop	= {execution-loop | call-sequence}, parametric-iterations
call-sequence	= {called-operation, parametric-iterations}*
parametric-iterations	= {[parameter-range], nmb-iterations}*
critical-section	= call-sequence
called-operation	= operation-id, interface-id, arg-data
arg-data	= passed-argument*, returned-argument*
passed-argument	= arg-name, arg-type, data-amount
returned-argument	= arg-name, arg-type, data-amount
data-amount	= {[parameter-range], element-size}*, {[parameter-range], nmb-elements}*
parameter-id	= GUID
parameter-type	= "boolean" | "byte" | "short" | "integer" | "long" | "double" | "float"
parameter-name	= CHAR*
min-possible-value	= standard-nmb-type
max-possible-value	= standard-nmb-type
default-value	= standard-nmb-type
min-value	= standard-nmb-type
max-value	= standard-nmb-type
mutex-mode	= BOOLEAN
standard-nmb-type	= BOOLEAN | BYTE | SHORT | INTEGER | LONG | DOUBLE | FLOAT
nmb-iterations	= non-negative-number
non-negative-number	= {"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"}*
positive-number	= {"1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"}*

Figure 5.4: *The component behaviour model in the EBNF notation.*

The EBNF-based specification of the behaviour model is given in Fig. 5.4. The interpretation of the model is as follows. The `behaviour-model` contains behaviour descriptions of each operation implemented by the component services (`operation-behaviour`). Also, the model defines a set of `parameters` on which the behaviour of operations depends. Each `parameter` is distinguished by `parameter-name`, `parameter-type` and `parameter-id`. Besides

this, the parameter element defines the `instantiation-constraints` for parameter values, such as the `min-possible-value`, `max-possible-value` and optional `default-value`. The latter three elements pre-define the range of values, in which an architect can instantiate each parameter in a scenario model. The parameter-value instantiation for each scenario is discussed in Section 5.4.2.

The `operation-behaviour` is described by a reference to an operation (`operation-id`), `behaviour-description` and `mutex-mode`. The boolean `mutex-mode` element is a synchronization primitive, which defines whether execution of an operation can be preempted by a scheduler.

The `behaviour-description` is defined by a parameterized specification of a `call-graph`. A *call graph* is a sequence of operations invoked during the execution of the specified operation. In other words, the call graph represents the internal behaviour of an operation in terms of communication with interfaces of other services.

It is important to emphasize that the concept of an interface allows specifying such communication dependencies of operations already at the phase of component development. A component developer specifies for a service a required interface that needs to be bound by a software architect to a provided interface of the same `interface-type` of another service. A component developer does not need to know the exact service implementation to which his required interface will be bound, because it can be any service with a provided interface of that `interface-type`. Therefore, the description of a call graph for each operation of a service is able to specify the called operations of other interfaces, before it is known to which other services this service is bound.

The call graph may depend on input parameters. A component developer may specify a number of parameter-dependent call graphs for each operation. This is done by the `parameter-range` element associated with each `call-graph`. The models of individual components use value ranges to reduce the modeling work. Each `parameter-range` defines the `min-value` and `max-value`. For instance, for each value range of a specific parameter, a component developer can define a specific call graph. The decision on the number and granularity of ranges to specify is left to the component developer.

Please note, that a `call-graph` can be specified such that it depends on multiple parameters. In the system design phase, the exact parameter values are instantiated (defined) by a system architect and, based on these values, a specific `call-graph` is selected by a model synthesizer.

Next, we explain the structure of the `call-graph` in more detail. It contains zero or more `execution-loop` elements and zero or more `critical-section` elements. The `critical-section` specifies synchronization constraints for a scheduler. In detail, it defines a sequence of called operations that cannot be preempted by a scheduler. The `execution-loop` is a recursive definition of `call-sequence`s. The `execution-loop` enables modeling of loops. A

parameter-dependent number of loops can be specified via the `parametric-iterations` construct.

The `call-sequence` specifies a sequence of calls to operations of other interfaces (`called-operation`) during the execution of the modeled operation. If a `called-operation` is invoked a number of times in a loop, a parameter-dependent number of iterations can be specified by the `parametric-iterations` element.

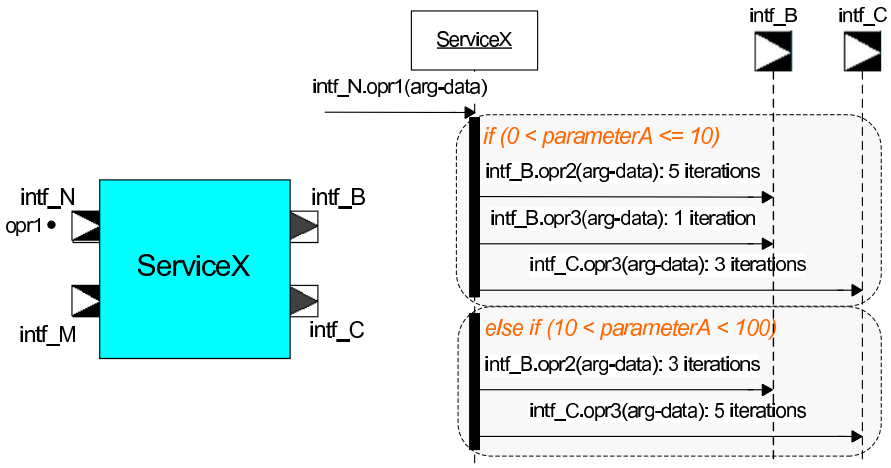

Figure 5.5: *MSC-based example of an operation behaviour description in a component behaviour model.*

The `called-operation` contains a reference to the operation identifier `operation-id` and to the identifier of the interface that offers this operation (`interface-id`). Besides this, data about the operation arguments `arg-data` can be modeled for each called operation. The `arg-data` may contain a number of `passed-arguments` and `returned-arguments`. These two elements allow specifying the amount of data that is transferred bi-directionally during the operation call (`data-amount`). The `passed-argument` defines a number of bits sent to the called operation at invocation time, while the `returned-argument` defines the number of bits that are sent back to the calling operation upon the completion of the `called-operation`. The `data-amount` element, represents the amount of the data to be sent and received. It can be specified in a parameter-dependent way. The `element-size` defines the argument size in terms of the number of bits (e.g. an argument of type `integer` contains 32 bits), while the `nmb-elements` enables to model array-type arguments.

Simple example of behaviour model

A simple graphical representation of a `behaviour-description` of an operation

is given in terms of a message sequence chart (MSC) in Fig. 5.5. The chart shows the modeled behaviour of the operation `opr1` implemented by the service `serviceX` and offered by the interface `intf_N`. The `serviceX` is dependent on two required interfaces `intf_B` and `intf_C`. The MSC of the operation behaviour shows that, once `opr1` is invoked, its behaviour depends on a value of `parameterA`. The exact value should be defined later on at the composition time and specified in a scenario model representing the assembly. At the component modeling time, the operation behaviour needs to be specified for all possible values of the parameter. In the example, the values are grouped into two ranges, namely from 0 to 10 inclusive, and from 10 to 100.

For the first value range, the operation `call-graph` includes three `called-operation` entities. First, the `opr2` is iteratively called five times via the interface `intf_B`. Second, the `opr3` is called once via the same `intf-B`. Finally, `opr1` invokes three times the operation `opr3` via the interface `intf_C`. Upon the return of the control flow from the `opr3`, the `opr1` completes its execution and returns the control flow to the entity that invoked the `opr1`.

For the second value range, the `opr1` invokes two operations sequentially. First, the `intf_B.opr2` is called three times, and then the `intf_C.opr3` is invoked in five iterations.

More detailed examples of a behaviour model for real applications can be found in Sections 9.3.1 and 9.4.2.

End of example

At the component assembly phase, the `call-graph`s of the operations from a sequence can be composed. Once the static software architecture, representing service instances and their bindings, is defined, the `call-graph`s of individual operations can be composed into a task `call-graph` containing the information about the sequence of executed operations of the constituent services. The composition algorithms are described in Section 5.5 in more detail.

The behaviour model specification is the responsibility of a component developer. The data for the model can be obtained either through top-down design or by source-code analysis of the component services.

Let us briefly discuss a few *limitations of the behaviour model* presented here. The granularity of modeling of service behaviour is limited to the level of operations. This level may appear to be too coarse-grained for obtaining very accurate behaviour predictions. However, our case studies revealed that this level is sufficient for assessing performance with acceptable accuracy (see Chapter 9). An extension would be to model behaviour on the level of processor instructions and employ a cycle-true simulator. Yet, this would make models large and difficult to specify, as well as time-consuming to simulate.

We have also limited modeling of programming constructions to parameter-dependent loops and excluded all types of conditional statements. This allows keeping models relatively simple and easy to construct.

5.3.3 Component Process Model

The component process model aims to enable modeling of components that create and execute their own operating system processes or threads. We call such components active. In more detail, the process model specifies parameter-dependent behaviour of the processes created and executed within the boundaries of services of individual active components.

Component Process Model

process-model	= process-in-service*, parameter*
process-in-service	= process-name, process-id, service-id, process-properties
process-properties	= virtual-operation, activation, deactivation, periodicity
virtual-operation	= operation-id, behaviour-description, mutex-mode
activation	= "by-default" | by-operation
deactivation	= "by-default" | by-operation
by-operation	= "by-operation", operation-id, provided-interface
periodicity	= periodic | "while-do-loop"
periodic	= "periodic", period
period	= positive-number
process-name	= CHAR*
process-id	= GUID

Figure 5.6: *The component process model in the EBNF notation.*

The EBNF definition of the component process model is depicted in Fig. 5.6. The following paragraphs explain important aspects of the process model. The `process-model` contains behaviour specifications of each `process-in-service` started and executed by one of the services of the modeled component. Besides this, the root defines a set of `parameters`, on which the behaviour of the component's processes depends. The explanation of the `parameter` element is given in Section 5.3.2. The `process-in-service` specifies a number of important `process-properties`, such as `virtual-operation`, type of the process `activation` and `deactivation`, as well as `periodicity` of a process.

The `virtual-operation` is introduced to specify the behaviour details of a source code of the process. For that, it includes the `behaviour-description`, the structure of which is given in Section 5.3.2. The `virtual-operation` should be provided with its own `operation-id` and modeled in the component resource model (see Section 5.3.2), where its hardware resource claims are to be specified.

The process `activation` specifies if the process is activated upon the start of the service execution or upon invocation of an operation offered by one of the `provided-interfaces` of the service. This enables modeling of different implementations of a process activation. The same modeling applies for the `deactivation` element. The `periodicity` element allows modeling two types

of implementation of process loop iterations. The `while-do-loop` element is used when a process loop iterates upon completion of a previous loop, while the `periodic` element is used when process loops are executed periodically with a certain `period`.

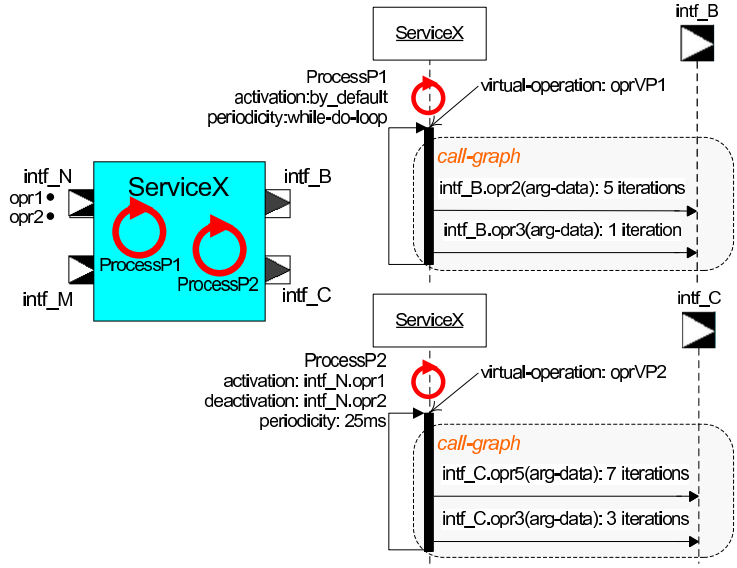

Figure 5.7: *MSC-based example of process-in-service description in a component process model.*

Simple example of process model
Fig. 5.7 depicts an MSC-based graphical example of two processes `ProcessP1` and `ProcessP2` run by `ServiceX`. The `ProcessP1` is activated by default and iterates in a `while-do-loop`. The process execution is emulated by virtual operation `oprVP1`. The `behaviour-description` of the virtual operation contains a call graph, in which a sequence of invocations of operations provided by interfaces of other services is performed. The `ProcessP2` is activated upon invocation of operation `intfN_opr1` implemented by `ServiceX`, and deactivated upon invocation of operation `intfN_opr2`. The process is defined as `periodic` where the process loop is iteratively repeated every 25 ms. A virtual operation `oprVP2` emulates the source code execution of the process. A detailed example of a process model for real application can be found in Section 9.4.2.
End of example

A component developer is responsible for specifying the process model. The model data can be obtained either by top-down design or by a source code analysis of processes of the component services.

5.3.4 Component Resource Model

The objective of a component resource model is to specify the hardware resource requirements of each operation implemented by the component. The requirements can be specified on a parameter-dependent basis.

Component Resource Model

resource-model	= operation-resource-use*, parameter*
operation-resource-use	= operation-id, cpu-use, memory-use
cpu-use	= {processor-type, {[parameter-range], cpu-claim}*}*
memory-use	= {[parameter-range], {memory-allocation, memory-release}}*
processor-type	= "RISC" | "CISC" | "VLIW"
cpu-claim	= mean-case, best-case, worst-case
average-case	= nmb-cycles
best-case	= nmb-cycles
worst-case	= nmb-cycles
memory-allocation	= nmb-bits
memory-release	= nmb-bits
nmb-cycles	= non-negative-number
nmb-bits	= non-negative-number

Figure 5.8: *The component resource model in the EBNF notation.*

The EBNF definition of the component resource model is shown in Fig. 5.8. The resource model contains `operation-resource-use` and `parameter` elements. The `parameter` element is explained in Section 5.3.2, while the former includes specifications of operation processor usage (`cpu-use`) and memory usage (`memory-use`).

The `cpu-use` element allows specifying the parameter-dependent `cpu-claim` of an operation. The `cpu-claim` defines the number of processing cycles (`nmb-cycles`) that an operation requires to complete its execution. The operation's `cpu-claim` value is to be defined for one of the specific `processor-type`, namely RISC, CISC or VLIW. To further empower the performance analysis, the `cpu-claim` data should be profiled and specified for three types of cases: `mean-case`, `best-case` and `worst-case`.

The `memory-use` is defined by a cumulative amount of memory allocated (`memory-allocation`) and released (`memory-release`) at operation execution.

The resource model specification is the responsibility of a component developer. The model data can be obtained either by estimation at the design time or by profiling the component execution at the stand-alone mode.

Simple example of resource model

Fig. 5.9 shows an example of a specification of RISC-processor operation claims resulting from `opr1-opr3`. Here, the profiling of `ServiceX` results in the fol-

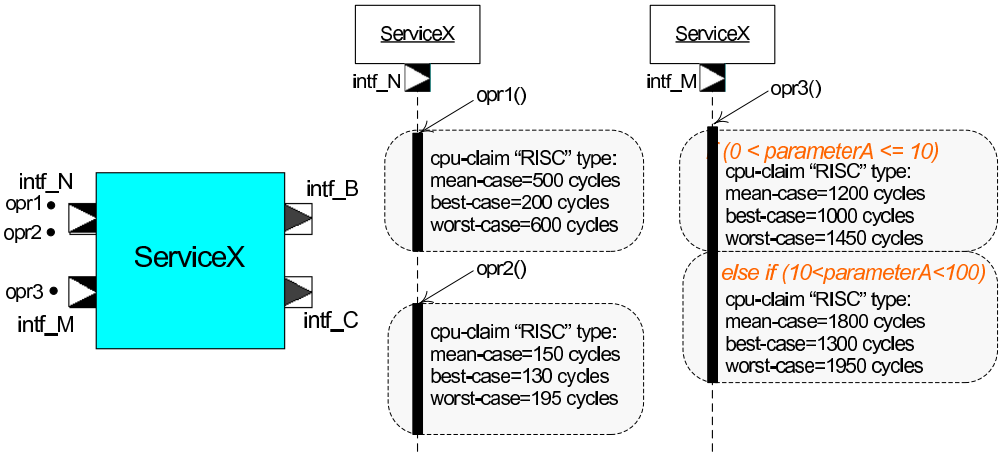

Figure 5.9: *An example of operation CPU claim specification in a component resource model.*

lowing values. The mean-case execution of `opr1` has been averaged to 500 processing cycles, while the worst-case and best-case execution traces took 600 and 200 cycles, respectively. The parameter-dependent processor usage data of `opr3` contains the following values. If the value of `parameterA` is within the 0-10 range, then the worst-case and best-case execution traces take 1,450 and 1,000 cycles, respectively. If `parameterA` value is in the range between 11 and 100, then the worst-case and best-case CPU claims are equal to 1,950 and 1,300 cycles, respectively. Detailed examples of a resource model for real applications can be found in Sections 9.3.1 and 9.4.2.
End of example

It is important to note that the resource model enables specification of operation processor requirements independently from the actual frequency of a processor. That is achieved by specifying claims in terms of processing cycles. This has the advantage that at the system design phase, the resource model is applicable to any deployment scheme, where the processor, on which the service providing the operation is deployed, can have an arbitrary frequency specification.

5.3.5 Hardware Performance Model

The hardware performance model targets at specification of hardware (processor, memory and bus) characteristics at a high-level of abstraction. The EBNF definition of the model is given in Fig. 5.10. The model falls into the three

categories: `processor-model`, `memory-model` and `bus-model`.

Hardware Performance Model

hw-performance-model	= {processor-model | memory-model | bus-model}, hardware-model-id
processor-model	= processor-id, processor-type, cpu-frequency
processor-type	= "RISC" | "CISC" | "VLIW"
memory-model	= memory-id, memory-type, size-bytes
memory-type	= "RAM" | "DRAM" | SDRAM" | "Flash"
bus-model	= bus-id, bus-type, bandwidth-kbits-sec, protocol-type
bus-type	= "PCI" | "I2C" | "PI" | "CAN" | "AMBA" | "Thinnet"
protocol-type	= "TDMA" | "CDMA" | "StaticPriority" | "FIFO" | "Lottery" | "CSMA/CD"
hardware-model-id	= GUID
processor-id	= GUID
memory-id	= GUID
bus-id	= GUID
cpu-frequency	= positive-number
size-bytes	= positive-number
bandwidth-kbits-sec	= positive-number

Figure 5.10: *The hardware performance model in the EBNF notation.*

The `processor-model` specifies the `processor-type` and `cpu-frequency` in terms of a number of processing cycles per second. The `processor-type` defines the type of processor instruction-set: RISC, CISC or VLIW. The `memory-model` specifies `memory-type` and its capacity in bytes (`size-bytes`). The memory types include (but are not limited to) the following: RAM, DRAM, SDRAM and Flash. The `bus-model` defines `bus-type`, `protocol-type` and bus bandwidth (`bandwidth-kbits-sec`).

The data for the hardware performance models can be obtained from measurements or from supplier data-sheets. The hardware type-set can be further extended when needed.

Modeling limitations. We have deliberately kept the hardware performance model simple, avoiding such details as processor caching strategies, bus and memory-access arbiters. Despite that this simplicity brings a certain level of inaccuracy in performance predictions, it enables rapid performance analysis in return, which becomes important when an architect has to consider multiple architectural alternatives within short period of time.

5.4 Architectural Models

In contrast to the model of software components and hardware IP blocks given in Section 5.3, the architectural models are generated at the architecture design phase of the DeepCompass framework (see Fig. 4.3). These models are *scenario-*, *deployment-* and *executable system models*. An architect can define

the scenario and deployment models in two ways: (a) by constructing architectural compositions via the CARAT graphical editors (see Chapter 7), with further generation of the models from these compositions and, (b) manually, by creating the XML files of the models. The executable system model is obtained by an automated synthesis of the scenario, deployment and individual component/hardware models.

5.4.1 Discussion on Scenario and Deployment Models Forming Architecture Alternatives

The purpose of this section is to clarify how the scenario and deployment models are used to generate various architecture alternatives. The contents is therefore explanatory by nature. In the next sections we provide a detailed specification of the scenario, deployment and executable system models. The objective of the *scenario model* is to specify both service composition (logical software architecture) and usage scenarios of an architecture alternative. For each alternative, different scenarios may differ only in input trigger specifications and should employ the same logical software architecture. An architect may define as many usage scenarios for a system as needed. In Fig. 5.11, we have used two scenarios for each architecture alternative, where each scenario contains three different triggers, while specifying the same service composition.

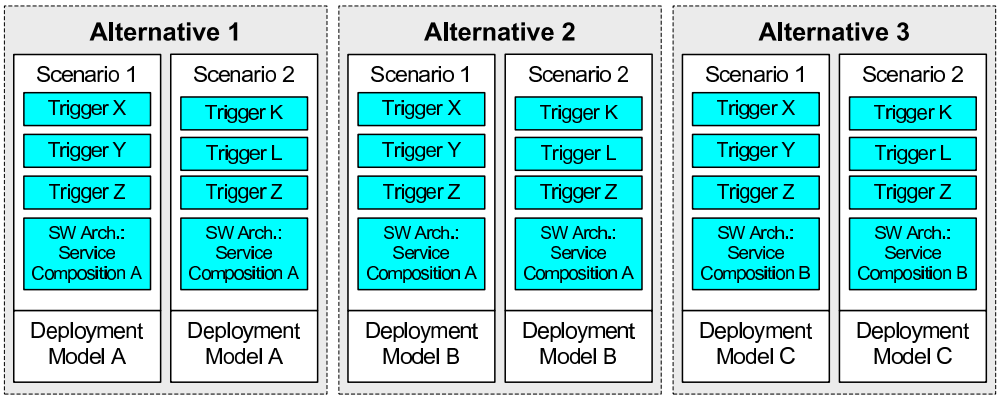

Figure 5.11: *Relations between architecture alternatives and scenarios, as well as their service compositions and triggers.*

A *deployment model* together with a set of scenario models forms an architectural specification of an alternative. The objective of the deployment model is to specify the hardware architecture and the deployment of the service instances (specified in a service composition of a scenario model) on the hardware nodes. Only one deployment model can be specified for an alternative.

The deployment schemes and the service compositions may vary per alternative. In Fig. 5.11, Alternatives 1 and 2 define the same Service Composition A, while Alternative 3 defines Service Composition B. Also, all three alternatives use different deployments specified in Deployment Models 1-3.

The scenario-based method assigns one important constraint on the usage of the triggers within scenario models. The identified set of triggers, forming a number of scenarios, should be applied for all alternatives in the same manner.

The synthesis of the executable system model is performed for each pair of scenario and deployment models. Therefore, an alternative may have a number of executable system models. The simulation of these models provides predictions on the performance properties. If several models provide predictions for a specific performance property, the worst-case prediction result should be taken for validation against corresponding performance requirements.

The following sections present the EBNF specifications of the scenario, deployment and executable system models along with graphical examples for each of the model types.

5.4.2 Scenario Model

Fig. 5.12 depicts the EBNF specification of the `scenario-model`. The four main elements of the model define the `assembly` of services, `triggers`, `process-instances` and `instantiated-parameters`.

The `assembly` describes the software architecture in terms of service composition. It includes a set of `service-instances` and `bindings` that connect these service instances.

The `trigger` specifies environmental-, platform- or user-based stimuli, that trigger an execution of a task. The `trigger` includes a specification of a trigger `periodicity` and defines an operation invoked upon arrival of a triggering event: `triggered-operation`. Besides this, an architect may optionally define a `priority` for the task created by the trigger and a `deadline` constraint for completion of the task. The trigger `periodicity` defines if the triggering events arrive in a `periodic`, `aperiodic` or `sporadic` way. A periodic trigger is characterized by its `period`, while a sporadic trigger is characterized by its minimal inter-arrival time (`min-interarrival-time`). An aperiodic trigger features three timing attributes: `burst-intraperiod`, `burst-interperiod` and `burst-length`.

The `triggered-operation` specifies which operation is invoked by the trigger and includes `operation-id`, `service-inst-id` as well as the provided port offering this operation `pp-name`.

In the `instantiated-parameter` element, an architect is supposed to assign values to all the parameters defined in the behaviour-, process- and resource models. The element includes the reference to the parameter (`parameter-id`) and `instantiated-value` fields. Based on the assigned parameter val-

Scenario Model

scenario-model	= assembly, trigger*, process-instance*, instantiated-parameter*, scn-model-id
assembly	= service-instance*, binding*
service-instance	= service-id, service-inst-name, service-inst-id, is-mapped-on
is-mapped-on	= processor-inst-id | memory-inst-id
binding	= binding-id, {service-inst-id, pp-name}[1], {service-inst-id, rp-name}[1]
trigger	= trigger-id, periodicity, triggered-operation*, [deadline], [priority]
periodicity	= periodic | sporadic | aperiodic
periodic	= "periodic", period, [offset], [jitter],
sporadic	= "sporadic", min-interarrival-time, [offset]
aperiodic	= "aperiodic", burst-interperiod, burst-intraperiod, burst-length, [offset]
triggered-operation	= operation-id, service-inst-id, pp-name
instantiated-parameter	= parameter-id, instantiated-value
process-instance	= process-id, service-inst-id, [deadline], [priority]
instantiated-value	= standard-type1
service-inst-name	= CHAR*
scenario-model-id	= GUID
binding-id	= GUID
trigger-id	= GUID
service-inst-id	= GUID
offset	= positive-number
jitter	= positive-number
min-interarrival-time	= positive-number
burst-interperiod	= positive-number
burst-intraperiod	= positive-number
burst-length	= positive-number

Figure 5.12: *Scenario model in the EBNF notation.*

ues, the corresponding call graphs and resource-usage data will be selected later by the CARAT toolkit for automated composition and performance analysis.

The `process-instance` element points to the service instance starting and executing this process instance: (`service-inst-id`). The `process-instance` allows to optionally specify `deadline` and `priority` of the task generated by the process instance.

Simple example of scenario model
Fig. 5.13 depicts a simple graphical example of a scenario model. In the example, three service instances A, B and C are composed by binding their interfaces K, L, M and N. Two triggers 1 and 2 with assigned periods, offsets and deadlines are triggering the operations `Intf_N.opr1()` `Intf_M.opr2()`, respectively. The parameters X,Y and Z are instantiated to values defined by an architect specifically for this scenario. A detailed example of a scenario model for a real application can be found in Section 9.3.3.
End of example

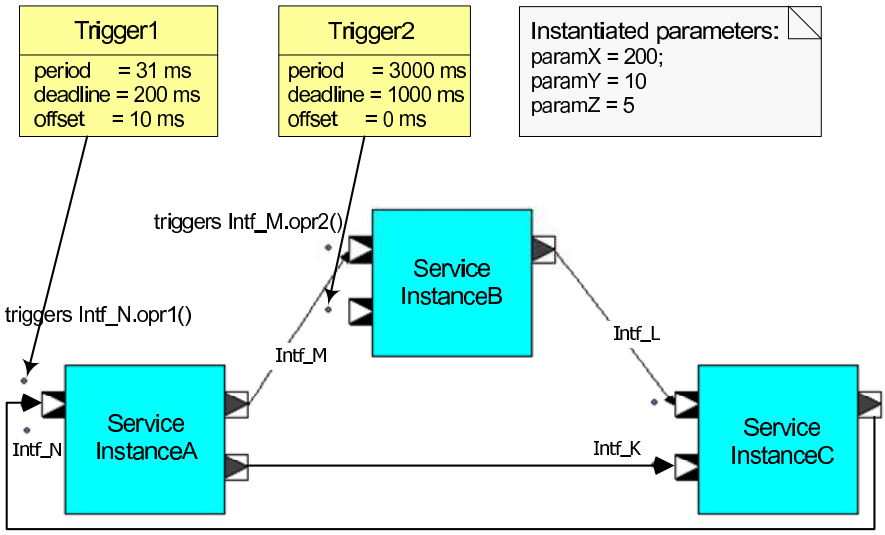

Figure 5.13: *Graphical example of a scenario model.*

Scenario identification is not a trivial process. An architect should (a) carefully analyze all possible incoming events and parameter configurations of a system, (b) extract the critical configurations that might hinder the performance requirements and (c) model them in a number of scenario models. We have conducted an empirical study on the issue of scenario identification and selection. The findings from the study are combined into scenario identification guidelines and are reported in Section 8.3.3. We present a brief outline of the guidelines for critical scenario identification. The performance requirements, system overload conditions, pick-load situations and high-frequency hardware interrupts are the source for performance problems in a system, and therefore, each of these factors should be modeled as a separate scenario.

5.4.3 Deployment Model

The *deployment model* aims at specifying the hardware architecture and the mapping of software service instances (defined in the scenario model) onto hardware nodes.

Fig. 5.14 presents the EBNF specification of the `deployment-model`. The model contains specification of zero ore more `processor`, `global-memory` and `bus` hardware IP blocks. The `processor` includes a reference to the processor specification in the hardware performance model: (`processor-id`). Besides this, it defines the processor instance `processor-inst-id` and availability of

Deployment Model

deployment-model	= processor*, global-memory*, bus*, depl-model-id
bus	= bus-id, bus-inst-id
processor	= processor-id, processor-inst-id, service-inst-id*, local-memory*, bus-inst-id*
global-memory	= memory-id, memory-inst-id, service-inst-id*, bus-inst-id*
local-memory	= memory-id, memory-inst-id, service-inst-id*
depl-model-id	= GUID
processor-inst-id	= GUID
memory-inst-id	= GUID
bus-inst-id	= GUID

Figure 5.14: *Deployment model in the EBNF notation.*

`local-memory` blocks on the processor. Finally, the `processor` specifies which service instances are deployed on the processor instance (`service-inst-id`), and which bus instances (`bus-inst-id`) are connected to the processor instance.

The `global-memory` contains a reference to the memory block specification in the hardware performance model (`memory-id`) and defines the global memory instance `memory-inst-id`. Besides this, `global-memory` specifies which service instances are deployed on the memory block (`service-inst-id`), and which bus instances (`bus-inst-id`) are connected to this global memory instance. As mentioned in Section 5.3.1, only the services representing buffers can be mapped on a memory block. These services are characterized by the element `is-buffer` set to `true`.

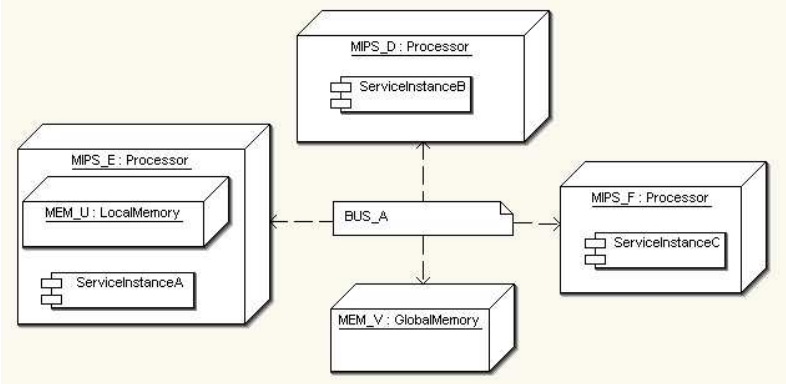

Figure 5.15: *Graphical example of a deployment model.*

Simple example of deployment model
Fig. 5.15 shows a simple example of a deployment model. The model contains three processor instances `MIPS_E`, `MIPS_D` and `MIPS_F`, on which the Service instances A, B and C are deployed. The `MIPS_E` processor contains local memory block `MEM_U`. The global memory block `MEM_V` and the processors are connected via bus instance `BUS_A`.
End of example

The CARAT toolkit provides an architect the necessary graphical means to design the deployment diagrams, checks the scenario and deployment diagrams for completeness and consistency, and generates the scenario and deployment models.

5.4.4 Executable System Model

The *executable system model* is synthesized from the above-described architectural models and individual component models. The synthesis is fully automated by the CARAT toolkit. The synthesized executable system model represents a *task execution architecture* that contains parameter-dependent data on the tasks running in the designed system, and the allocation of these tasks within the software and onto the hardware architectures. More specifically, a synthesized *task* is a schedulable entity with a detailed description of: (1) a triggering entity (trigger or process) that initiates the task, (2) the task's period, deadline, priority, offset and jitter, (3) a sequence of operations and data transfers executed within a task and (4) processor and communication load imposed by each operation and data transfer within a task. The obtained tasks are suitable for different types of performance analysis (simulation, schedulability or hybrid), resulting in predictions for behaviour and timing properties of a system.

Let us now describe the relation between the synthesized *tasks*, the *triggers* (obtained from the scenario model) and the *processes* (obtained from the process models of components). Both the triggers and processes serve as an input for identification of tasks in the system. The processes are general OS-related entities, while a task is a scheduling entity. Processes and trigger events create (fire) task instances [21]. These task instances are scheduled by the CARAT Simulator tool. For example, in an Anti-lock Braking System (ABS), a triggering event, informing that the wheels are locked, initiates a task instance that executes anti-locking functionality. Another example is a data-streaming process which executes the following three operations within a loop: read data from ports, process data and write data to ports. Here, one loop iteration creates a task instance, executing the functionality of these operations.

The parameters of a trigger are used as the parameters for the task in-

stances initiated by this trigger. For instance, a trigger deadline specifies the deadline for completion of each task instance. The processes specify periodicity parameters for initiated task instances. In case a process is a part of a multimedia streaming chain which contains a number of active components running the processes in parallel, the process deadline may be neglected. Instead, an architect can specify a deadline for a complete processing loop of a video frame (such processing loop typically starts from a read-frame operation and ends with a render-frame operation). In this case, a designer specifies a loop deadline by means of the CARAT toolkit. In order to maintain a one-to-one relation between the deadline of a process and the deadline of a corresponding task instance, the process model does not support fork/join specification of a process.

The tasks identification is a rigid and automated procedure. An architect may influence the set of tasks running in the system by specifying the scenario model. Once the scenario model is defined, the task set cannot be changed and optimized as proposed in [32].

Executable System Model

executable-system-model	= constituent-models, execution-architecture
constituent-models	= scn-model-id, depl-model-id, hardware-model-id, component-id*
execution-architecture	= executable-task*
executable-task	= created-by, execution-sequence, task-name, task-id
created-by	= {process-id, service-inst-id} | trigger-id
execution-sequence	= {executed-operation | data-transfer}*
executed-operation	= operation-id, service-inst-id, pp-name, processor-inst-id, execution-time
execution-time	= min-exec-time-ms, avr-exec-time-ms, max-exec-time-ms
data-transfer	= from-operation, to-operation, bus-inst-id, direction, transfer-amount-bits
from-operation	= operation-id, service-inst-id, pp-name
to-operation	= operation-id, service-inst-id, pp-name
direction	= "forward" | "return"
min-exec-time-ms	= DOUBLE
avr-exec-time-ms	= DOUBLE
max-exec-time-ms	= DOUBLE
transfer-amount-bits	= DOUBLE
task-name	= CHAR*
task-id	= GUID

Figure 5.16: *Executable system model in the EBNF notation.*

Fig. 5.16 depicts the executable system model in EBNF notation. The model consists of the `execution-architecture` and `constituent-models` elements. The latter element contains references to all architectural- and component models involved. The `execution-architecture` specifies a set of `executable-tasks` obtained at the model synthesis. Each `executable-task` is associated with its `task-id` and `task-name`. Also, `executable-task` defines

its `execution-sequence` and the entity that creates the task: `created-by`. The `created-by` element references either to `trigger-id` if the task is created by a trigger, or to a pair `process-id-service-inst-id` if the task is created by a process.

The `execution-sequence` data is composed out of behavior- and process models of individual components and defines a sequence of multiple `executed-operations` and `data-transfers`. In other words, each element in the task `execution-sequence` can be either an **operation** executed on a certain processor, or a **data-transfer** performed via a bus once an operation passes or returns some arguments to an operation residing on a different processor.

An individual `executed-operation` is defined by a reference to a specific operation and service instance in the software architecture (`operation-id` and `service-inst-id`). Besides this, the `executed-operation` includes an operation execution time value (`execution-time-ms`) computed from the data given in corresponding resource and hardware performance models. The value is obtained for the specific processor `processor-inst-id`, on which the service instance implementing the operation is mapped in the deployment model.

The `data-transfer` element defines the operation's `from-operation` and `to-operation`, exchanging the arguments data via bus `bus-inst-id`. The constituent element `direction` specifies if the data transfer is performed for a forward or return operation calls. Furthermore, `data-transfer` includes a value for the transferred amount of bits (`transfer-amount-bits`) computed from the `arg-data` specified in the corresponding behaviour or process model.

The following section defines the algorithms for synthesis of the executable system model.

5.5 Synthesis of the Executable System Model

This section explains the algorithms for the synthesis of the architectural and component models into the executable system model. The algorithms represented here are implemented and validated in the CARAT toolkit (see Chapter 7). Fig. 5.17 gives an overview of the synthesis process. The process involves all the models described in previous sections and includes the following four steps:

- *Step 1: Initialization of executable tasks.* Identification of all `trigger` and `process-in-service` elements from the scenario- and component process models. Generation of the corresponding `executable-task` objects.
- *Step 2: Synthesis of task call graphs.* For each `executable-task`, a call graph is recursively composed based on the individual `call-graphs` of operations involved.

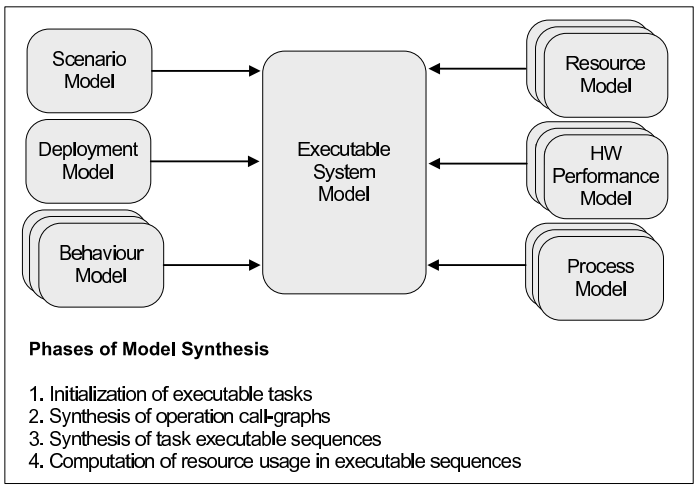

Figure 5.17: *Synthesis of the executable system model out of architectural and component models.*

- *Step 3: Synthesis of task execution sequences.* For each `executable-task`, the `executed-operation` and `data-transfer` elements are identified and stored in the task `execution-sequence`.
- *Step 4: Computation of resource consumption within execution sequences.* For each element in each `execution-sequence` either `execution-time` or `transfer-amount-bits` is computed, based on data from corresponding deployment, resource and hardware performance models.

The following sections specify these four steps of the synthesis algorithm in detail. For each step, a synthesis algorithm is given in pseudocode notation. The outcome of each step is represented as an extended MSC diagram. Moreover, each MSC diagram provides an example of a possible result.

5.5.1 Step 1. Initialization of executable tasks

The goal of the first synthesis step is to identify and initialize all possible tasks that are executed in the specific scenario. Stream-based and control-based systems feature different types of task initialization mechanisms. In control-based systems, tasks are initialized by triggers, while in stream-based systems, tasks are initialized by processes running inside active components. For this reason, the algorithm incorporates two types of task initialization mechanisms: trigger-based and process-based.

Fig. 5.18 shows that the data for this step is fetched from the scenario model and the process models of the components involved with the composition. From

the scenario model, the algorithm fetches the tasks which are fired by triggers (trigger-based). From the process component models, the algorithm fetches the tasks which are fired by the processes created in active components (process-based).

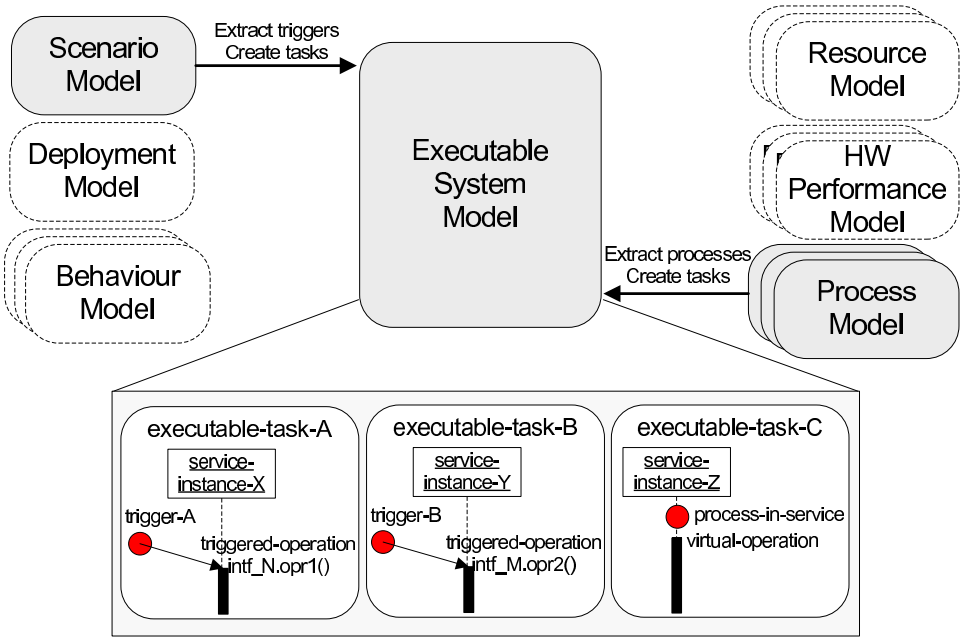

Figure 5.18: *Step 1. Model view of the task initialization. The enlarged view at the bottom is a possible example of the result of Step 1.*

Fig. 5.19 depicts two algorithms of the task initialization in pseudocode: `InitializeTriggerTasks` and `InitializeProcessTasks`. The former algorithm scans through all `triggers` in the scenario model and identifies the `triggered-operations` of each trigger. The following actions establish the trigger-based tasks of the future executable system model.

1. Initialize an `executable-task` for each triggered operation and generate a unique `task-id`.
2. Set up the `created-by` property of the task and link it to the `trigger-id`.
3. Initialize `execution-sequence` and the first `execution-operation` of the task.
4. Set the `triggered-operation` as the first `execution-operation` in the task `execution-sequence`.

The `InitializeProcessTasks` algorithm performs initialization of tasks fired by the `process-instances` of those process models, which services are

```
1 PROCEDURE InitializeTriggerTasks()
2  FOR each trigger in the scenario-model
3     FOR each triggered-operation of the trigger
4        INITIALIZE executable-task and GENERATE task-id
5        SET executable-task.created-by.trigger-id as the trigger-id
6        INITIALIZE executable-task.execution-sequence
7        INITIALIZE executed-operation
8        SET executed-operation.operation-id as the triggered-operation.operation-id
9        SET executed-operation.service-inst-id as the triggered-operation.service-inst-id
10       INSERT the executed-operation as a first item to executable-task.execution-sequence
11    ENDFOR
12  ENDFOR
13 END PROCEDURE

1 PROCEDURE InitializeProcessTasks()
2  FOR each process-instance in the scenario-model
3     INITIALIZE executable-task and GENERATE task-id
4     SET executable-task.created-by.process-id as the process-instance.process-id
5     SET executable-task.created-by.service-inst-id as the process-parent service-inst-id
6     REQUEST user to define deadline and priority for the process-instance
7     INITIALIZE executable-task.execution-sequence
8     INITIALIZE executed-operation
9     SET executed-operation.operation-id as the process-in-service.virtual-operation.operation-id
10    SET executed-operation.service-inst-id as the process-in-service.service-inst-id
11    INSERT the executed-operation as a first item to executable-task.execution-sequence
12  ENDFOR
13 END PROCEDURE
```

Figure 5.19: *Step 1. Pseudocode specification of the task initialization algorithms InitializeTriggerTasks and InitializeProcessTasks.*

instantiated in the scenario model. For each `process-instance`, the procedure generates a process-based task by the following sequence of steps.

1. Initialize an `executable-task` for each `process-instance` and assign a unique `task-id`.
2. Set up the `created-by` property of the task and link it to the `process-id`.
3. Ask an architect to define the optional `deadline` and `priority` properties for the `process-instance`
4. Initialize `execution-sequence` and first `execution-operation` of the task.
5. Set the `virtual-operation` of the process as the first `execution-operation` in the task `execution-sequence`.

The described step generates an initial task structure of the future executable system model. The structure defines a set of executable tasks (see Fig. 5.18) and, for each of them, reveals the following three properties: (a)

which entity triggers the task and; (b) which operation is first to execute once the task instance is fired; (c) the periodicity of the job (task instance) firing and the job completion requirements (deadline). For example, in Fig. 5.18, Trigger A invokes the triggered operation `opr1` of Service instance X, and by this, forms the initial definition of Executable task A. This definition will be further supplemented in the following steps.

5.5.2 Step 2. Synthesis of task call-graphs

The aim of the task call-graph synthesis is to compose individual call graphs of operations into a call graph of an executable task. The synthesis is performed on the basis of behaviour and process models of individual services and their binding specification fetched from the scenario model. The synthesis procedure is similar to the method recently presented in [98].

For each task, the algorithm reconstructs a sequence of operations to be executed upon triggering of a task instance. The call graphs of individual operations, given in the behaviour and process models of corresponding components, serve as building blocks for the synthesis. Note that, the synthesis is only possible once a component assembly is defined in a scenario model, containing specification of service instances and bindings between them. Fig. 5.20 depicts the models used at this step and an example of an obtained task call-graph in terms of an MSC diagram.

The call-graph synthesis algorithm is based on the pre-order traversal principle [9]. The synthesis algorithm is shown in Fig. 5.21. It contains root procedure `SynthesizeCallGraphs` and traversing procedure `PreorderTraversal`.

The following paragraphs explain important aspects of the algorithm. For each `executable-task` identified in the previous step, the `SynthesizeCallGraphs` algorithm performs the following actions.

1. Fetch `behaviour-description` of the first `executed-operation` in the `execution-sequence`. For each task the first `executed-operation` was created in the previous step of task initialization.
2. Identify the `call-graph` that applies to the `instantiated-values` of the `parameters` specified in the `behaviour-description`. Here the algorithm resolves the parametric dependencies of the operation behaviour, defined in the behaviour or process models of the component providing the operation. The parameter values are given by the scenario model.
3. Initialize `curr-operation-list`, which will store the traversed sequence of the task operations. Add the first `executed-operation` to the `curr-operation-list`.
4. Set the counter `position-in-curr-list`, specifying the current position of an operation in the `curr-operation-list` to 0.
5. Execute the `PreorderTraversal` procedure that recursively traverses the `call-graphs` of operations, thus, synthesizing the complete sequence

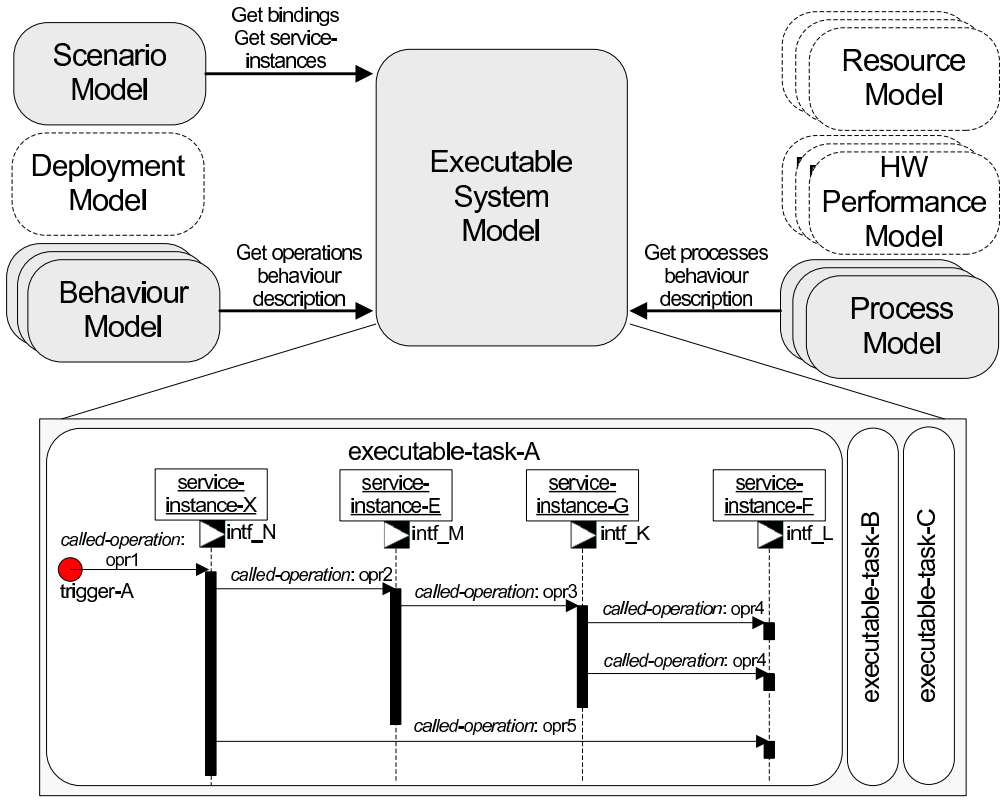

Figure 5.20: *Step 2. Model view of the synthesis of the operation call graphs.*

of operations to be executed by the task. The following arguments are passed to the procedure: `call-graph`, `curr-operation-list` and `position-in-current-list`. The procedure returns the traversal results in the `curr-operation-list` argument.

6. Initialize empty `final-operation-list` of the task and store the resulting `curr-operation-list` into it.

The internal `PreorderTraversal` recursive procedure accepts two input arguments (`IN`), i.e. `call-graph` and `position-in-curr-list`, as well as one input-output argument (`INOUT`), `curr-operation-list`. The procedure executes another recursive algorithm `FlatenCallGraph`, that flattens the `execution-loop` elements in the current `call-graph` converting them into a local operation sequence. The resulting operation sequence is returned by the procedure in the `tmp-operation-list`. The `FlatenCallGraph` is a preparatory procedure for traversal of call graphs and shown in Fig. 5.22.

The obtained `tmp-operation-list` is inserted into the `curr-operation-`

```
1 PROCEDURE SynthesizeCallGraphs()
2  FOR each executable-task identified in the scenario-model
3     GET the first executed-operation from the executable-task.execution-sequence
4     GET behaviour-description of the executed-operation.operation-id in its behaviour-model
5     IDENTIFY parameters defining the call-graph of the behaviour-description
6     IDENTIFY instantiated-value for each parameter from the scenario-model
7     IDENTIFY which call-graph applies for these instantiated-values
8     INITIALIZE empty curr-operation-list
9     ADD the first executed-operation to the curr-operation-list
10    SET position-in-curr-list to 0
11    EXECUTE PreoderTraversal(call-graph, curr-operation-list, position-in-curr-list)
12    INITIALIZE empty executable-task.final-called-operation-list
13    APPEND curr-operation-list to the executable-task.final-operation-list
14  ENDFOR
15 ENDPROCEDURE

1 PROCEDURE PreoderTraversal(IN call-graph, INOUT curr-operation-list, IN position-in-curr-list)
2  INITIALIZE empty tmp-operation-list
3  EXECUTE FlatenCallGraph(call-graph, tmp-operation-list)
4  INSERT the obtained tmp-operation-list into curr-operation-list at (position-in-curr-list + 1)
5  FOR each called-operation in the tmp-operation-list
6     GET the position of the called-operation in the curr-operation-list
7     SET position-in-curr-list to obtained position number
8     GET operation-id of the called-operation
9     GET behaviour-description of the operation-id in this behaviour-model
10    IDENTIFY parameters defining the call-graph of the behaviour-description
11    IDENTIFY instantiated-value for each parameter from the scenario-model
12    IDENTIFY which call-graph applies for these instantiated-values
13    IF (call-graph != empty) THEN
14       EXECUTE PreoderTraversal(call-graph, curr-operation-list, position-in-curr-list)
15    ENDIF
16  ENDFOR
17 ENDPROCEDURE
```

Figure 5.21: *Step 2. Pseudocode specification of the synthesis of the operation call graphs.*

`list` in the position `(position-in-curr-list+1)`. Then, for each of the `called-operation` in the `temp-operation-list`, the following set of actions is performed.

1. Obtain the position of the `called-operation` in the updated `curr-operation-list` and store the value as `position-in-current-list`.
2. Get `operation-id` of the `called-operation` and find its `behaviour-description` in the corresponding behaviour or process model.
3. Identify which `call-graph` applies to the `instantiated-values` of the `parameters`, on which the operation behaviour is depending.

```
PROCEDURE FlatenCallGraph(IN call-graph, INOUT called-operation-list)
  INITIALIZE called-operation-list as empty
  FOR each execution-loop in the first level of the call-graph
    EXECUTE TraverseExecutionLoop(execution-loop, called-operation-list)
  ENDFOR
END PROCEDURE

PROCEDURE TraverseExecutionLoop(IN execution-loop, INOUT called-operation-list)
  IF the execution-loop is a call-sequence THEN
    FOR each called-operation in the call-sequence
      IDENTIFY parameters defining the parametric-iterations for the call-sequence
      IDENTIFY instantiated-value for each parameter from the scenario-model
      IDENTIFY which nmb-iterations of the called-operation applies for these instantiated-values
      REPEAT for nmb-iterations
        APPEND the called-operation to the called-operation-list
      ENDREPEAT
    ENDFOR
  ELSEIF the execution-loop is nested execution-loop THEN
    IDENTIFY parameters defining the parametric-iterations for the nested execution-loop
    IDENTIFY instantiated-value for each parameter from the scenario-model
    IDENTIFY which nmb-iterations of the nested execution-loop applies for these instantiated-values
    REPEAT for nmb-iterations
      EXECUTE TraverseExecutionLoop(execution-loop, called-operation-list)
    ENDREPEAT
  ENDIF
END PROCEDURE
```

Figure 5.22: *Step 2. Pseudocode specification of the pre-order traversal the operation call graphs.*

4. If the `call-graph` contains at least one `called-operation`, then perform the recursive `PreorderTraversal` procedure for this `call-graph`. The current `curr-operation-list` and `position-in-curr-list` values are passed to the procedure. As a result, the procedure updates the `curr-operation-list` and returns it.

Concluding, the recursive traversal algorithm walks through `call-graphs` of all operations involved in the task execution and builds up the sequence of these operations. As a result, the operations are ordered in the way they are executed by the task.

A graphical example of the preorder traversal is shown in Fig. 5.23. The operation `OprA` is the root of the tree, which, in the context of our approach, means that it is either a triggered operation or a virtual operation of a process instance. The call graph of `OprA` includes two called operations `OprB` and `OprC`. The latter operations have their own call graphs. The `OprB`, in its turn, calls `OprE` and `OprF`, while the `OprC` invokes the following three operations `OprX`, `OprY` and `OprZ`. The last three operations are leafs, except for `OprZ`, which calls two other leaf operations `OprK` and `OprL`. A leaf operation is an operation

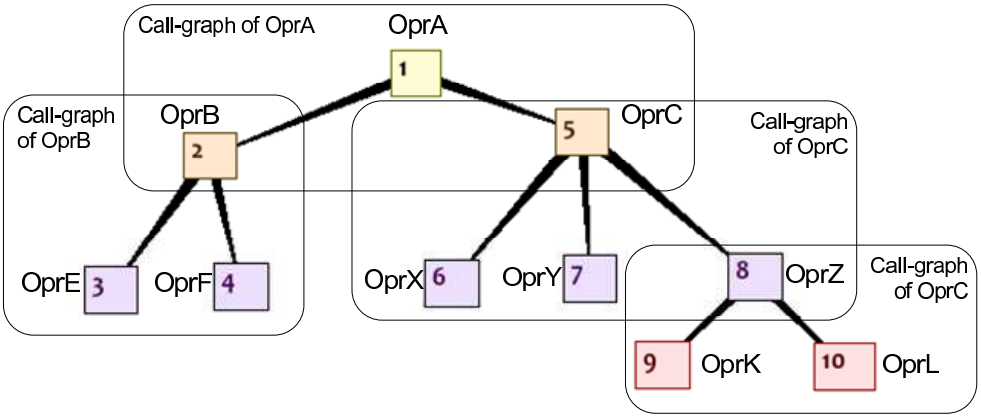

Figure 5.23: *Example of the preorder traversal of the operation call graphs.*

that does not invoke any other operations while being executed. The preorder traversal results in the following sequence of operations to be executed once the task instance is fired: `OprA, OprB, OprE, OprF, OprC, OprX, OprY, OprZ, OprK, OprL`.

Example of a task call-graphs synthesis.
Fig. 5.20 depicts an example of the synthesis results in terms of an MSC diagram. Here, the algorithm identified a sequence of operations invoked within the task `executable-task-A`, triggered by `trigger-A`. The traversal shows that the service instances `X, E, F, G` are involved in the task execution. The `opr1` is a triggered operation, while the `final-operation-list` contains operations `intf_N.opr1`, `intf_M.opr2`, `intf_K.opr3`, `intf_L.opr4` (twice) and `intf_L.opr5`.
End of example

The algorithm of the call-graph synthesis features the important compositionality property of our behaviour and process models. Being specified at the component development phase, these models are applicable to an arbitrary future composition of services at a system design phase (with a constraint on proper binding of provided and required interfaces of these services).

5.5.3 Step 3. Synthesis of task-execution sequences

The goal of this third step is to identify and construct a sequence of computational and communication activities within a task. The need for this step comes from the following reasoning. In the previous step, of a task call-graph

synthesis we obtained a sequence of operations performed by a task upon a triggering event. The execution of these operations imposes a certain load on processing nodes. These operations also exchange data by passing arguments to each other. The transfer of these arguments imposes a certain load on communication hardware resources, if the arguments are passed from one processor to another. To make accurate modeling of these communication activities, we need to identify and integrate them into the executable system model.

A *computational activity* is an execution of an operation that imposes a load on a processing node. A *communication activity* is a data transfer between two operations executing on different processors, thereby imposing a load on a bus/network. A data transfer between two operations is performed when a called operation accepts some input arguments or when it returns some output arguments. As shown in Fig. 5.24, the algorithm collects data from the deployment model, as well as from the behaviour and process models of the involved components.

Fig. 5.25 represents the synthesis of task execution sequences in pseudocode. The algorithm works as follows. For each of the `called-operation`s in the `final-operation-list` of a task, the algorithm identifies if the operation is invoked with passed or returned arguments. Should this be the case, then the algorithm identifies if the service instances executing the `called-operation` and `calling-operation` are deployed on different processing nodes (`IsDataTransferViaBus` procedure). If this is true, a `data-transfer` element is instantiated representing an inter-processor communication activity in the task. The `bus-inst-id` property of the data transfer is assigned to the `bus-inst-id` of the communication line connecting the two processing nodes. As a result, for each task an `execution-sequence` is generated, containing a mixed sequence of `operation-execution` and `data-transfer` actions. The sequence orders the actions as they will be performed during an execution of a task instance.

Example of the resulting task-execution sequences.
An example of such a synthesis result is depicted in Fig. 5.24 in terms of an MSC diagram. Here, the service instances are shown deployed on specific Processor instances A and B. Besides this, Bus C connects the two processors. The diagram depicts the data transfer activities (*data-transfer: via Bus C*) assigned to communication between Service instances E and G deployed on Processor instances A and B. The remaining operation invocations are performed within the same processing node, so that no `data-transfer` actions are assigned for them. The detailed synthesis example on the real applications is given in Sections 9.2.4 and 9.4.6.
End of example

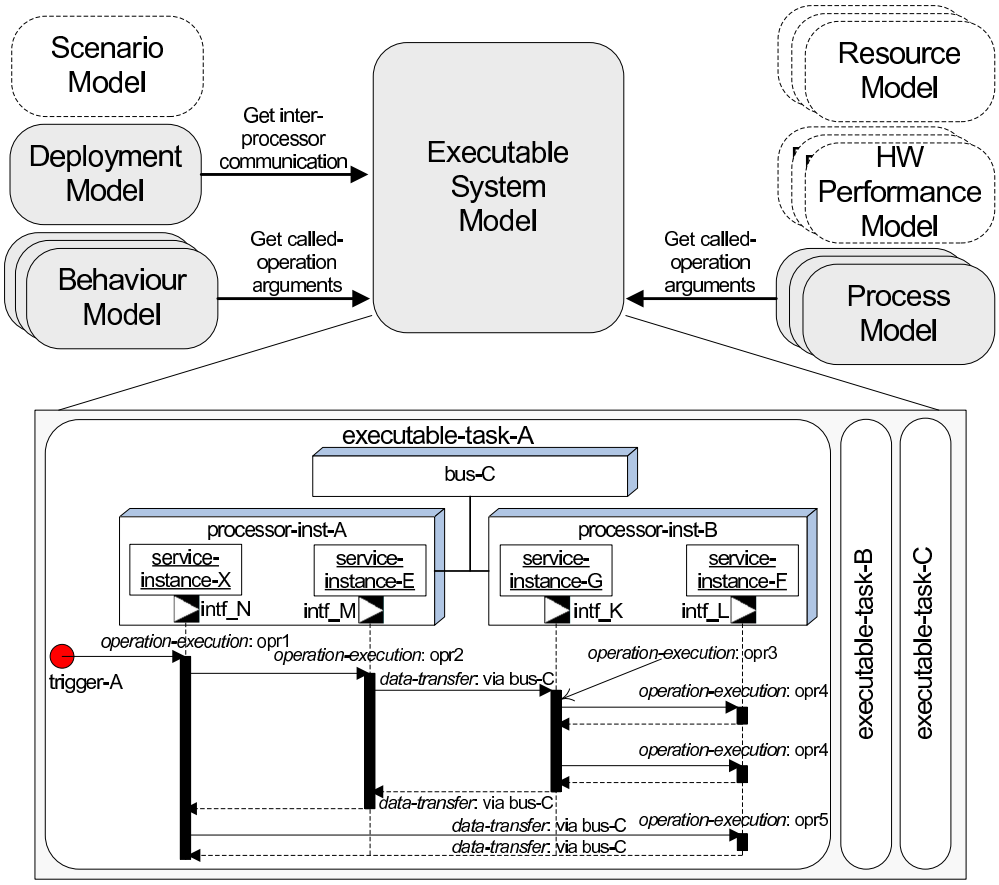

Figure 5.24: *Step 3. Model view of the synthesis of the tasks executable sequence. The enlarged view is a possible example of Step 3.*

5.5.4 Step 4. Computation of resource consumption within execution sequences

The last step of the synthesis computes the hardware resource usage of each action within an execution sequence. Fig. 5.26 shows that this involves the resource models of components, hardware performance models and the deployment model.

The detailed algorithm of this step is given in Fig. 5.27. Here, the `Compute-ResourceUsage` procedure goes through the `execution-sequence` of a task and computes the resource usage for each action in the sequence. For an action of type `operation-execution`, the procedure identifies the `cpu-frequency` of the processor which executes the operation, and the procedure then calculates

```
PROCEDURE SynthesizeTaskExecutionSequence()
  FOR each executable-task
    FOR each called-operation starting from second in the executable-task.final-operation-list
      IDENTIFY the calling-operation that invokes the called-operation
      IF (called-operation has passed-arguments) THEN
        EXECUTE IsDataTransferViaBus(calling-operation, called-operation, bus)
        IF (bus != null) THEN
          INITALIZE data-transfer object
          SET data-transfer.from-operation as the calling-operation
          SET data-transfer.to-operation as the called-operation
              SET data-transfer.direction as "forward"
          SET data-transfer.bus-inst-id as id of the found bus instance
          APPEND the data-transfer to the executable-task.execution-sequence
        ENDIF
        INITIALIZE operation-execution object
        SET operation-execution.operation-id as called-operation.operation-id
        SET operation-execution.service-inst-id as called-operation.service-inst-id
        APPEND the operation-execution to the executable-task.execution-sequence
      ENDIF
      IF (called-operation has returned-arguments) THEN
        EXECUTE IsDataTransferViaBus(called-operation, calling-operation, bus)
        IF (bus != null) THEN
          INITALIZE data-transfer object
          SET data-transfer.from-operation as the called-operation
          SET data-transfer.to-operation as the calling-operation
          SET data-transfer.direction as "return"
          SET data-transfer.bus-inst-id as id of the found bus instance
          APPEND the data-transfer to the executable-task.execution-sequence
        ENDIF
      ENDIF
    ENDFOR
  ENDFOR
ENDPROCEDURE

PROCEDURE IsDataTransferViaBus(IN from-operation, IN to-operation, OUT bus)
  IDENTIFY from deployment-model the processor on which the service-instance executing the from-
operation is deployed
  SET the from-operation.processor-inst-id
  IDENTIFY from deployment-model the processor on which the service-instance executing the to-
operation is deployed
  SET the to-operation.processor-inst-id
   IF (from-operation.processor-inst-id == to-operation.processor-inst-id) THEN
      IDENTIFY the bus connecting these two processors
   ENDIF
ENDPROCEDURE
```

Figure 5.25: *Step 3. Pseudocode specification of the synthesis of the tasks execution sequence.*

the minimum-, average- and maximum operation execution time, based on the `cpu-claim` values given in the corresponding resource model.

For an action of type `data-transfer`, the procedure accumulates the `transfer-amount-bits` that are sent via the corresponding bus either upon an in-

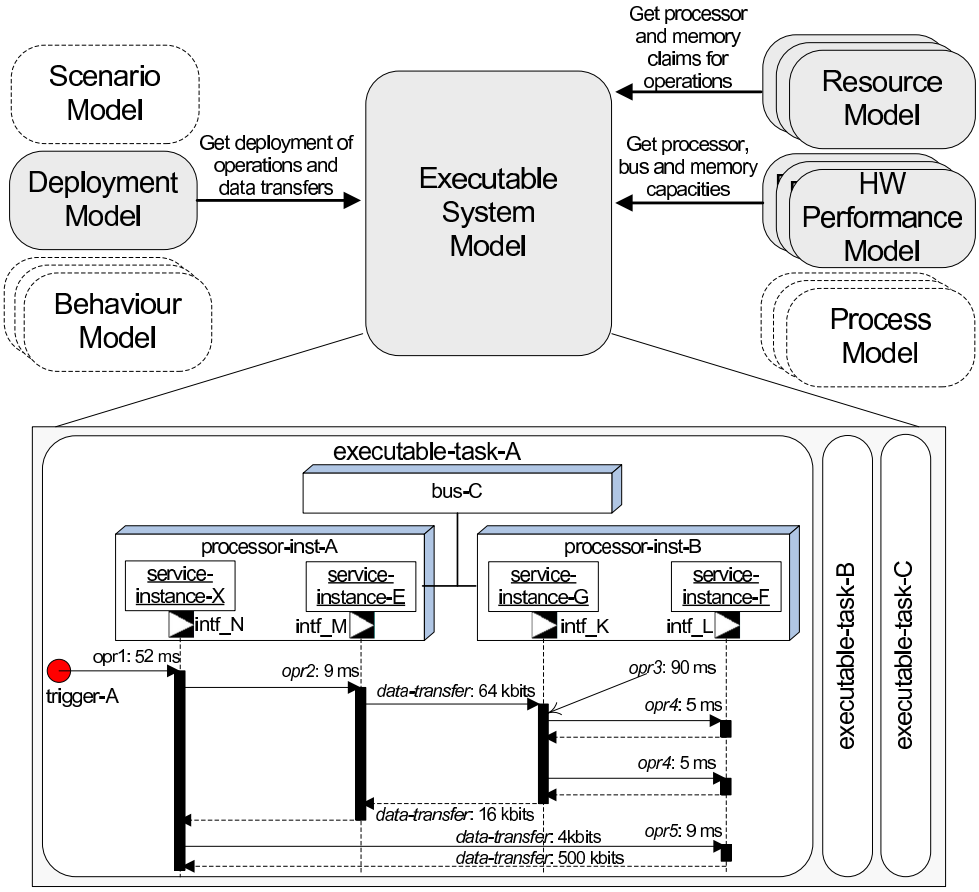

Figure 5.26: *Step 4. Computation of resource usage for operations and data transfers in executable sequences of tasks. Model view.*

vocation of an operation with `passed-arguments`, or upon a return call of an operation with `returned-arguments`.

Example of the resulting resource consumption.
The MSC-based result of this last synthesis step is shown in Fig. 5.26. We obtain a so-called annotated MSC diagram, where for each action an execution time or a transfer size is specified. For instance, `opr1` is executed for 52 ms on the Processor instance A. The data transfer between `Opr2` and `Opr3` contains 64 kbits of data during the invocation of `Opr3` and 16 kbits of data upon the return call.
End of example

```
PROCEDURE ComputeExecutionSequenceResourceUsage()
  FOR each executable-task
    FOR each object in the executable-task.execution-sequence
      IF (the object is of type operation-execution) THEN
        IDENTIFY processor-id deploying the operation-execution.service-inst-id from the
deployment-model
        IDENTIFY processor-type of the processor-id in its processor-model
          IDENTIFY cpu-frequency of the processor-id in its processor-model
        IDENTIFY operation-execution.operation-resource-use from its resource-model
        IDENTIFY operation-resource-use.cpu-claim for the processor-type in the resource-model
        ComputeTime (cpu-claim.best-case, cpu-frequency, operation-execution.min-exec-time-ms)
        ComputeTime (cpu-claim.mean-case, cpu-frequency, operation-execution.avr-exec-time-ms)
        ComputeTime (cpu-claim.worst-case, cpu-frequency, operation-execution.max-exec-time-ms)
      ELSEIF (the object is of type data-transfer) THEN
        IF (data-transfer.direction == "forward") THEN
          GET data-transfer.to-operation.arg-data from behaviour-model of the to-operation
          FOR each passed-argument in the arg-data
            ComputeTransferAmount (passed-argument.data-amount, curr-transfer-amount-bits)
            SET data-transfer.transfer-amount-bits += curr-transfer-amount-bits
          ENDFOR
        ELSEIF (data-transfer.direction == "return") THEN
          GET data-transfer.from-operation.arg-data from behaviour-model of the from-operation
          FOR each returned-argument in the arg-data
            ComputeTransferAmount (returned-argument.data-amount, curr-transfer-amount-bits)
            SET data-transfer.transfer-amount-bits += curr-transfer-amount-bits
          ENDFOR
        ENDIF
      ENFIF
    ENDFOR
  ENDFOR
ENDPROCEDURE

PROCEDURE ComputeTime (IN nmb-cycles, IN cpu-frequency, OUT nmb-ms)
  SET nmb-ms = (1000 x nmb-cycles) / cpu-frequency
ENDPROCEDURE

PROCEDURE ComputeTransferAmount (IN data-amount, OUT transfer-amount-bits)
  IDENTIFY parameters defining the element-size and nmb-elements of the data-amount
  IDENTIFY instantiated-value for each parameter from the scenario-model
  IDENTIFY values of element-size and nmb-elements that apply for these instantiated-values
    SET transfer-amount-bits = element-size x nmb-elements
ENDPROCEDURE
```

Figure 5.27: *Step 4. Computation in pseudocode of resource usage for operations and data transfers in execution sequences of tasks.*

The resource-usage computation step finalizes the synthesis of the executable system model. The model specifies a set of tasks executing in a specific scenario. For each task, the triggering, behaviour and hardware resource usage are defined.

5.6 Performance Analysis of Exec. System Model

The obtained executable system model is a subject for performance analysis (PA). A number of PA approaches can be applied to the model. These approaches can be categorized in two types: static analysis and simulation-based techniques. Static analysis methods result in predicted worst- and best-case response times of each task and an overal CPU-, memory- and bus utilization for a specified scenario. Simulation-based techniques result in task-execution behaviour (patterns of execution), distribution of latencies for each task, and fine-grained (in time) resource utilization for a given scenario.

For the static analysis, Queuing Networks [38], Timed Petri Nets [22] or Real-Time Calculus [95] techniques can be deployed. The CARAT toolkit does not implement these techniques itself, but is able to convert the obtained system model into a specific input format for each technique.

A simulation-based analysis employs virtual schedulers that simulate the execution of the tasks specified in the system model for some period of time. The selection of a scheduling algorithm is dictated by the types of communication busses and operating system used for the designed system. The virtual schedulers for simulation, implemented by the CARAT toolkit, provide (but are not limited to) the following algorithms: Rate Monotonic (RM), Deadline Monotonic (DM), Earliest Deadline First (EDF), Time Division Multiple Access (TDMA), Round Robin (RR) and fair-use algorithms. Our virtual schedulers can be used for scheduling of both processing and communication resources.

5.6.1 Algorithm of the Simulation Scheduler

The simulation scheduler operates as follows. The tasks defined in the executable model serve as input for the scheduler. Besides this, an architect specifies the duration and time-slot for the simulation run. The time-unit is a configuration parameter, and represents the atomic unit of simulation. The time-unit is normally set to one millisecond or microsecond.

The simulation state-chart for task instances is presented in Fig. 5.28. At every time-unit, the scheduler examines the tasks in the executable system model with respect to appearance of new task instances. If a Task Instance (TI) appears (i.e. a job is fired), then the scheduler sets the instance to the `idle` state. Each task fires its instances according to their periodicity. For every time-unit, the scheduler checks the state of each task instance and, if needed, changes these states. Besides the above, the other states include `idle`, `ready-for-execution`, `blocked`, `operation-execution`, `ready-for-data-transfer`, `data-transfer` and `completed`. In the following paragraphs, we explain the states and transitions of task instances.

For each hardware node (processor or bus), the scheduler allocates a set

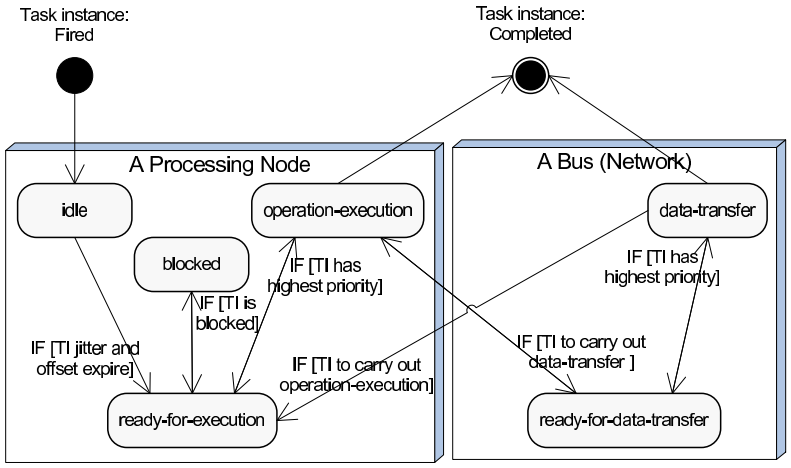

Figure 5.28: *State-chart of a task instance during a simulation.*

of queues to store task instances assigned to each node (see Fig. 5.29). Each queue is dedicated to task instances having a certain state. For each processing node, the scheduler runs queues of `idle`, `blocked` and `ready-for-execution` TIs. The `ready-for-execution` queue is re-ordered according to task instance priorities at every time slot. The task instance currently performing the `operation-execution` is stored in a separate container. For each bus, the scheduler has a queue of `ready-for-data-transfer` TIs and one container for a task instance performing `data-transfer`.

The conditions for changing the TI states are as follows. A task instance can be moved in the following cases.

- From an `idle` state to a `ready-for-execution` state when the TI jitter and offset are expired. According to the synthesis algorithm explained in Section 5.5, the first element of a task `execution-sequence` is always of the `executed-operation` type. Therefore, the task instance is put into the `ready-for-execution` queue of the processing node, on which the service, implementing this first operation, is deployed.
- From a `ready-for-execution` state to a `blocked` state if at least one task instance on this processing node steps in a mutexed mode or in a `critical-section`.
- From a `ready-for-execution` state to an `operation-execution` state if the TI has highest priority among other task instances on this processing node.
- From an `operation-execution` state to a `completed` state if the TI completes the execution of the operation and has no further actions in

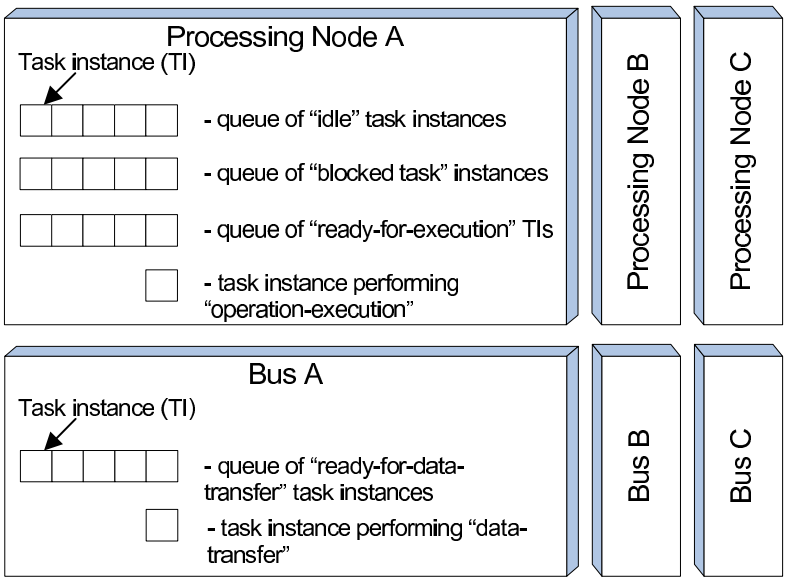

Figure 5.29: *Allocation of different queue types for processors and busses within a scheduler.*

its `execution-sequence`.

- From an `operation-execution` state to a `ready-for-data-transfer` state if the TI has completed execution of an `executed-operation` and the next object in its `execution-sequence` is of a type `data-transfer`. The task instance is put into a `ready-for-data-transfer` queue of the corresponding bus.
- From a `ready-for-data-transfer` state to a `data-transfer` state if the TI has highest priority among other task instances on this bus.
- From a `data-transfer` state to a `ready-for-execution` state if the TI has completed execution of the `data-transfer` and the next object in its `execution-sequence` is of a type `executed-operation`. The task instance is put into a `ready-for-execution` queue of a corresponding processing node.
- From a `data-transfer` state to a `completed` state if the TI completes the data transfer and has no further actions in its `execution-sequence`.

As we mentioned above, the scheduler performs priority-based ordering of the `ready-for-execution` and `ready-for-data-transfer` queues for each processor/bus at every time slot. The algorithms for prioritization are defined by a scheduling policy (RMA, DMA, EDF or TDMA) deployed for a hardware node. An architect defines the scheduling policies prior to simulation.

5.6.2 Performance results presentation

These schedulers provide various algorithms for processor- and bus-sharing arbitration. The designer inputs the executable system model to a virtual scheduler that should adhere to the scheduling policy of an OS of the designed system. A mixed or hierarchical scheduling can also be simulated. The simulation yields the utilization of resources and in the predicted task behaviour, latencies and number of missed deadlines. Fig. 5.30 depicts the CARAT simulation results from our validation case study for an MPEG-4 decoder.

The diagram shows the execution timelines of the three processors and the bus-load timeline. For each processor timeline, the tasks executing the operations of the services that are mapped on the processor, are shown. For each task instance, its initiation and completion times are given. Besides this, the diagram reflects the time-units when a task instance misses its deadline. The bus-load timeline represents the timed bus utilization done by the communicating operations in these three tasks. The statistics, generated from the simulation timelines, give the overall data on the predicted task properties and load of the resources.

5.7 Review of Assumptions and Limitations

The method presented in this chapter is based on a number of *assumptions*. Firstly, for every individual component and hardware IP block involved in a system design, the *set of models should be available* at the architecting phase. Creating the set of models imposes some overhead for a component developer, despite the high level of modeling abstraction. Secondly, *usage of scenarios requires that an architect has a good understanding* of the system-environment interaction aspects, and has some analytical skills in identifying the scenarios. The scenario identification challenge is addressed in Chapter 8, representing our industrial survey on scenario-based analysis techniques. Finally, we assume that the *deployment of software services onto hardware nodes is static*, i.e. a service is always executed on a specific processor. This assumption is valid due to the fact that a static deployment is used for real-time systems to achieve predictable behaviour.

A number of *limitations* that need further study are the following. At first, the behaviour model represents an abstraction of a source code, leaving out implementation details. This eases the assessment of the component and system behaviour, but *limits the modeling of detailed aspects of the source code*, like complex condition statements implemented inside a component operation. A second modeling limitation is that we model *processor, memory and bus characteristics at a coarse grain level*, thereby omitting caching and other peculiarities. This enables rapid simulation and performance predictions, but, may lead to some inaccuracy in results. Finally, the *simulation technique*

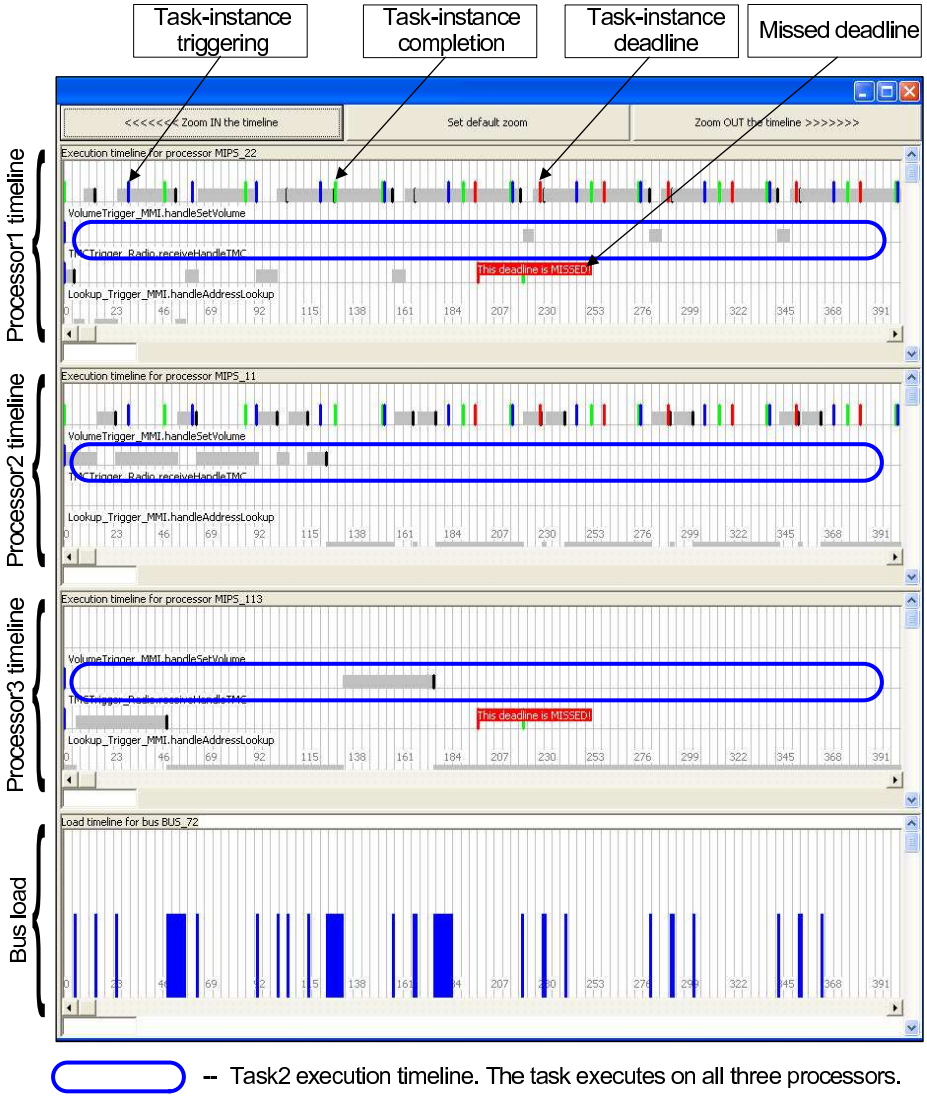

Figure 5.30: *Example of execution timeline for processing nodes, buffers and network obtained from simulation.*

does not guarantee finding boundary worst-case values for task latencies and resource utilization.

5.8 Conclusions

The scenario-based performance analysis method, described in this chapter, enables rapid prediction of behaviour and performance properties of component-based software systems. The targeted performance properties are task latencies, number of missed deadlines and worst-case processor/bus/memory loads. The identified behaviour properties are task blocking and interleaving, performance bottlenecks and distribution of processor/bus/memory loads.

The method is based on the following principles.

- Modeling of individual software and hardware components at a high abstraction level.
- Specification of system scenario models, which define stimuli that trigger task executions within a system.
- Automated synthesis of individual component models and architectural models into an executable system model, representing a specification of running tasks in a system.
- Simulation of the tasks, resulting in predicted performance properties.

Let us now conclude on the major achievements of this scenario-based method. The proposed assessment of performance properties only for a set of critical scenarios *substantially reduces the analysis time and efforts.* Indeed, scenarios allow avoiding analysis of the whole state space of a system and help focusing only on the critical system configurations and execution modes.

An important achievement of the method is that it enables *automated synthesis of individual component models* into a system-wide model *for any proper composition of an arbitrary set of components.* This automation, together with the rapid modeling and simulation facilities, makes this scenario-based approach suitable for designing multiple architectural alternatives with further comparison and optimization in a limited period of time.

Besides the above-mentioned advantages, the method features a number of other important benefits. Firstly, it allows modeling of both distributed systems and Systems-on-Chip (SoCs) with *heterogeneous hardware nodes.* Secondly, the method features *processor-independent modeling* of processor usage claims for service operations, which, in turn, allows the deployment of the software service on an arbitrary processor. Thirdly, the hardware usage and behaviour properties of a software component can be specified in a *parameter-dependent way.* This turns the component models into powerful means for representing complex functionality of a software component. Fourthly, the behaviour model provides facilities for specification of *task synchronization constraints* (mutexed operations and critical sections), which lead to fine predictions of task interleaving aspects. Finally, the approach allows an architect

to work both with *passive and active components*, thereby allowing to choose for an architectural style that better addresses the system requirements.

In the following chapters, we discuss various important aspects of the method. The performance prediction results obtained by the scenario-based method are used in the architecture optimization phase, described in Chapter 6. Chapter 7 describes the architecture of the CARAT software toolkit that implements the above-described model synthesis and simulation algorithms. In Chapter 8 we present our empirical case study on the industrial usage of scenarios for performance assessment. In that chapter, we justify our scenario-based method, based on the results from a number of interviews with leading industrial architects. Finally, Chapter 9 describes the case studies that we carried out in order to validate the DeepCompass framework and the scenario-based method. In this chapter, we present the MPEG-4 decoder, Car Radio Navigation system and JPEG application. We applied the DeepCompass process to design and predict the performance properties of the systems. The scenario-based method has been validated by comparison between the predicted and actually measured performance properties.

Chapter 6

Architecture Optimization

6.1 Introduction

6.1.1 Background on Optimization Methods

The development of time- and safety-critical systems requires architects to address a large number of non-functional requirements, such as performance, safety, availability, reliability and cost. One major difficulty is that these non-functional requirements conflict with each other. For instance, improving system performance often requires more powerful hardware nodes, which increases the production cost and power consumption. To construct a system that fulfills all its quality requirements is often not possible. As a consequence, an architect has to consider several design alternatives and identify a solution that satisfies most quality objectives, and where the optimal balance between different quality attributes is achieved. *Architecture optimization* is the process of generation of the design alternatives, then performing the trade-off analysis between the alternatives and finally selecting the optimal alternative.

Architecture optimization is an iterative process, which starts with the construction of an initial set of architectural alternatives/solutions. The following paragraph describes the three iterative phases used for architecture optimization.

1. *Evaluation of quality attributes of each architectural alternative.* This phase aims at obtaining the values of the quality attributes of an alternative, based on the architecture specifications. In Chapter 3, we have outlined the existing methods that can be used in this phase.

2. *Comparison of the alternatives: trade-off analysis.* In this phase, an architect compares the alternatives based on the obtained set of quality attributes. Due to the contradictory nature of the quality attributes, it is often not possible to identify which alternative outperforms all other alternatives. One alternative may score high in terms of performance, while another alternative may have low power consumption. Therefore, an architect identifies so-called *optimal architectures* that feature a balanced distribution of the quality attribute values. The Pareto-frontiers [74] and cost-function [94] techniques can be applied for identification of such optimal alternatives (also called non-dominated alternatives). A Pareto-optimal set contains individual solutions which are non-dominated by other solutions in the search space. (Solution A dominates Solution B if A outperforms B on every quality attribute). They represent a selection of trade-offs that can be presented to the designer to ultimately choose the best for a specific application. The found non-dominated alternatives may be used in (a) the next phase of architecture transformation, or (b) for selection of the final architecture. Moreover, the detailed trade-off analysis helps an architect to understand which design decisions impose the particular strengths/weaknesses of an alternative. As a result, an architect derives the guidelines or directions for further architecture transformations.

3. *Transformations of the alternatives or generation of the new alternatives.* The objective of this phase is to improve the quality attribute values, while preserving the system behaviour. The transformation techniques can be classified into two categories: manual and automated. The *manual* techniques use architectural patterns as a basis. They use pattern collections that describe a group of suitable transformations to improve one quality attribute (e.g., safety [42] and performance [29]). An example of such patterns is the integration of redundancy mechanisms in order to improve reliability [30]. Another quality improving measure is the reassignment of software components to other hardware nodes in order to avoid bottlenecks and to reduce the workload of a specific hardware node. At the opposite, the *automated* techniques use architecture specification and reasoning frameworks, to automatically generate a set of new alternatives based on the previous results. The methods for such automated design space exploration include the Monte Carlo Search, Simulated Annealing (SA), Tabu Search, Genetic Algorithms (GA), etc. In the following paragraphs, we discuss these exploration methods in more detail.

6.1.2 Scope of the Chapter

This chapter presents our ongoing research work in the field of architecture optimization. We survey state-of-the-art approaches in this field and identify their differences and commonalities (see Section 6.2). As a result of the survey, we construct an abstract architecture optimization method (see Section 6.3). In this method, we incorporate the beneficial points of every method and our own ideas on using an artificial intelligence for the architecture optimization. The abstract optimization method provides a skeleton for tailored instances of methods, thereby contributing to a common understanding of architecture optimization.

In order to characterize the abstract optimization method, the following benefits emerge to the foreground.

1. The method provides an integral approach, where all possible optimization phases (requirements and priorities definition, design and analysis of initial alternatives, identification of promising design-space directions and generation of new alternatives) are defined and handled in an iterative process.
2. The method provides one common model to be used as a skeleton to attach multiple analysis methods, leading to analysis integrity and reduction of development efforts.
3. The techniques for early identification of quality attributes and bottlenecks in the architecture alternatives lead to guidance for the generation of promising design alternatives, thereby avoiding brute-force search and reducing optimization complexity.

The DeepCompass framework can be considered as an instance of this abstract optimization method. The framework already includes most of the facilities for automated architecture optimization. It provides the functionality for rapid construction of a set of initial architectural alternatives. Furthermore, the framework enables *predictions* of multiple performance quality attributes of these alternatives. The predicted performance results can be used either for finding the optimal alternative, or for optimization of these alternatives with respect to better balancing their quality attribute values. The identification of optimal solutions is based on the Pareto-frontiers, while the further optimization of existing alternatives is based on the architectural patterns.

Fig. 6.1 depicts the context of this chapter. Our method for architecture optimization is a constituent part of the DeepCompass framework. The validation of the method is presented in Chapter 9, where we present our development case studies of the Car Radio Navigation system and JPEG application.

The chapter is structured as follows. In Section 6.2, we analyze the existing approaches for architecture optimization and, based on this analysis, present an abstract method for optimization in Section 6.3. In Section 6.4, the method

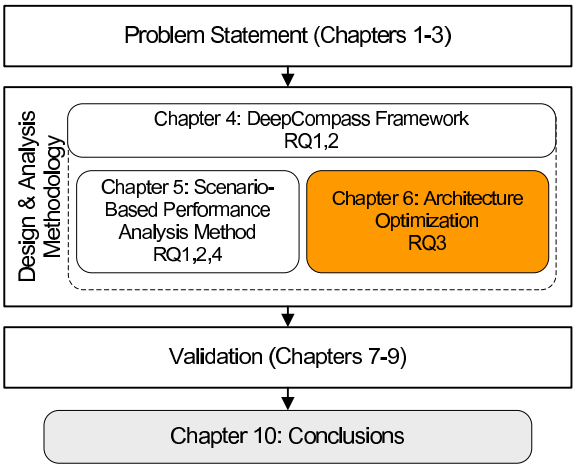

Figure 6.1: *Positioning of Chapter 6 in the thesis.*

is tailored for the context of the DeepCompass framework. Section 6.5 provides post-analysis of challenges in architecture optimization field. Section 6.6 concludes the chapter.

6.2 Architecture Optimization Approaches

This section outlines ten approaches from the literature over the last two decades, addressing the trade-off analysis and/or optimization on architectural design models. These approaches have focused on the optimization of multi-objective criteria, including safety, resource usage, timing performance, reliability and cost. The focus is on comparing the methods from five different viewpoints: (1) the optimization objectives pursued by each method; (2) the heuristic and meta-heuristic techniques used for searching in large design spaces and generation of alternatives; (3) the architectural models used for evaluation of quality attributes; (4) the techniques used for multi-objective optimization; and (5) the intended applications.

Approach 1. Nicholson [77] describes an approach that addresses reliability, resource usage, task latencies and cost criteria of the architecture. The approach involves two phases. The goal of the first phase is to find an optimal architecture topology which includes decisions about appropriate redundancies. The second phase aims at finding optimal deployment architectures, which specify how software components are mapped to real-time tasks and how these real-time tasks are then mapped to hardware nodes. Nicholson argues

that this distinction is necessary, since the two phases involve optimization problems with different complexities. As a result, Genetic Algorithms are applied in the first phase and Simulated Annealing is used as a search heuristic in the second phase. The used optimization parameters include worst-case response times (WCRT), reliability metrics (e.g. mean time to failure), resource usage metrics, and production costs. The overall approach is implemented in the tools X-Alloc and X-Topmeter and is used successfully on the case study of a computer-assisted braking system.

Approach 2. Thiele *et al.* [95, 105] present an approach for design space exploration and architecture optimization for Network Processor Architectures. The approach is based on models for packet processing tasks, a description of workload streams entering a system, and a specification of a feasible space of architectures including computation and communication resources. For each architectural alternative, the model data is analyzed with Real-Time Calculus [95], in order to evaluate performance. The designed alternatives are the subject for the multi-objective optimization performed by the SPEA2 framework [105]. This framework enables architecture optimization based on Genetic Algorithms.

Approach 3. Palermo *et al.* [80] introduce a framework for design space exploration specifically for parameterized embedded System-on-Chip (SoC) architectures. The framework aims at solving the power-consumption/response-time optimization problem for embedded devices. It uses the following three architecture generation and evaluation algorithms: Random Search Pareto (RSP), Pareto Simulated Annealing (PSA) and Pareto Reactive Tabu Search (PRTS). The authors claim that the combined use of these algorithms reduces search efforts by three orders of magnitude. The assessment of the relevant quality attributes is provided by simulation and dynamic profiling of the target systems. The framework is illustrated by the case study of a GSM encoding application.

Approach 4. Another technique for the design space exploration of embedded SoCs has been developed by Palesi *et al.* [35]. This technique focuses on the architecture optimization with respect to the power/latency trade-offs. For architecture generation and evaluation, it reuses the genetic algorithms implemented by the SPEA2 framework [105]. The instrumentation for architecture mutations is based on parameterizations of the hardware IP blocks.

Approach 5. A generic framework for architecture optimization has been proposed by Künzli *et al.* in [69]. This modular framework for design space exploration allows using various optimization techniques depending on the problem domain. The framework provides the following multi-objective opti-

Table 6.1: *Comparison of architecture optimization approaches against four important aspects.*

Method & Reference	**Application Area/Case Study**	**Optimization Approach/Strategy**	**Dependability Evaluation Models**	**Dependability Improving Measures**
Nicholson [77]	Safety-Critical, Real-time Systems/Computer Assisted Braking System	Multi-Objective Optimization/Genetic Algorithms and Simulated Annealing	Reliability Block Diagrams (RBD), Schedulability Analysis	N-Version Programming, Redundancy, Task Allocation
Thiele *et al.* [95]	Network Processor Architectures	Multi-Objective Optimization/Genetic Algorithms	Real-Time Calculus (RTC): Analytical approach employing workload and resource curves	Hardware Topology, SW/HW Mapping
Palermo *et al.* [80]	Embedded systems on SoC platforms/GSM encoding application	RSP, PRTS and PSA Algorithms/ power-vs-delay trade-off	Simulation and dynamic profiling	Hardware parameters: number and size of cache, ALU, multiplier and memory blocks
Palesi *et al.* [35]	Embedded systems on SoC platforms	Genetic Algorithms of SPEA2 framework/ power-vs-delay	Simulation	Hardware parameterizations
Künzli *et al.* [69]	Generic	Multi-Objective Optimization/Multiple randomized-search algorithms	Performance run-time profiling (used in case-study)	Task allocations, Hardware Topology, SW/HW Mapping, Parameters
Papadopoulos and Grante [82]	Safety-Critical Systems / Automotive design	Multi-Objective Optimization/Genetic Algorithms	Automatically constructed Fault Trees, Simple Weight and Cost Estimations	Component substitution and replication
Fredriksson *et al.* [32]	Real-time Systems/ Generic Case Study	Single-Objective Optimization/Genetic Algorithms	Schedulability Analysis, Analysis of the Memory Consumption	Component to Task Allocation
Grunske [43]	Mission-Critical Embedded Systems/Satelite Control Application	Multi-Objective Optimization/Evolutionary Algorithms	Reliability Block Diagrams (RBD), Simple Weight and Cost Estimations	Component Replication (Two Channel Redundancy, Triple Modular Redundancy)
Pimentel *et al.* [86]	Embedded systems on heterogeneous platforms	Multi-Objective Optimization	Trace transformations, Kahn Process Networks, Simulation	Task re-allocations, Hardware Parameters and Topology, SW/HW Mapping
Livolsi *et al.* [73]	Component-Based Systems/ Financial System Case Study	Multi-Objective Optimization/Evolutionary Algorithms		Hardware Parameters and Topology

mization techniques: black-box optimization, randomized search and problem-dependent search. The multi-objective evaluation module of the framework deploys the concept of Pareto-dominance. The framework is based on the PISA (Platform and programming language independent Interface for Search Algorithms) protocol that specifies a problem-independent interface between the search/selection strategies on one hand and the problem domain-specific estimation and variation operators on the other hand. The framework is illustrated by a simple application example where the design is optimized for obtaining the most efficient cache architecture.

Approach 6. Papadopoulos and Grante [82] describe an approach to multi-objective optimization of safety-critical automotive systems. The objective is to find optimal trade-offs among safety and reliability (treated as separate objectives) and cost in the design of such systems. A genetic algorithm that promotes population diversity [2] is used to progressively improve a Pareto set of non-dominated solutions that represent different trade-offs among the parameters of the optimization. The approach departs from earlier work in that the safety and reliability model (i.e. a set of system fault trees) is automatically synthesized from an engineering model that is augmented with information about component failures.

Approach 7. Fredriksson *et al.* [32] describe a single-objective optimization approach that is targeted towards hard real-time systems. The goal is to find an optimal allocation from components to tasks. Genetic algorithms are chosen as the optimization strategy, where each gene represents a component and contains a reference to the task it is assigned to. Each allocation produced by the GA is evaluated by a fitness function, which sums up memory consumption on the stack and CPU overhead. Both parameters are determined using a basic scheduling analysis algorithm, based on parameters such as worst-case execution times, or required stack usage, which are attached to each component in the system.

Approach 8. Grunske [43] describes a method that focuses on the improvement of reliability within given cost and weight constraints. The method uses a multi-objective optimization strategy that is implemented by a simple evolutionary algorithm which progressively improves a set of Pareto-optimal solutions. Reliability is evaluated using so-called Reliability Block Diagrams, which are generated separately for each function delivered by the system, based on the components that are needed to perform this function. The approach is illustrated on a case study of a satellite control system and two of its main functions.

Approach 9. Pimentel *et al.* [86] describe a SESAME framework for explo-

ration and optimization of embedded system architectures at multiple abstraction levels. The framework deploys both analytical and simulation methods for evaluation of quality attributes. Analytical methods are fast and work with high levels of abstraction, while simulation methods are more detailed and require detailed system specifications. The framework focuses on predicting performance-related quality attributes and exploration of heterogeneous multi-processing systems with respect to these quality attributes. The performance evaluation methods used are the following: Kahn Process Networks, trace transformations and simulation. The SPEA2 framework [105] is used for multi-objective architecture optimization. The approach is illustrated by a Motion-JPEG encoder case study.

Approach 10. Livolsi *et al.* [73] propose a guided architecture-based design optimization technique for component-based systems. The technique enables exploring possible architectures by repeatedly applying evolutions to initial architectures, with the quality attributes of each architecture being evaluated throughout. The found quality attributes guide the designer to the next iteration. An optimization cycle contains three modules: Corrector, Effector and Evaluator. The Corrector module takes initial architectures and system requirements as an input and creates generation guidelines for the Effector module. The latter applies evolutionary algorithms to generate a new population of architectural alternatives. The Evaluator module takes the new population, assesses its quality attributes and performs multi-objective evaluation using Pareto-optimal principles. The evaluation results are sent to the Corrector module for the next iteration. This optimization technique is supported by the ABACUS software toolkit.

An overview of the approaches is given in Table 6.1, where important aspects are listed, such as the application area, optimization strategy, quality evaluation models and quality improving measures.

The next section will present an abstract architecture optimization method, that extends the set of common elements found in the presented approaches here. The extension is in adding a design space analyzer defining guidelines for an evolution engine. Our method is open in the sense that we adopt techniques from the presented proposals wherever appropriate. However, our extension on guiding the architecture evolution can provide faster and higher quality results in the future, when new techniques such as artificial intelligence are incorporated.

6.3 Abstract Architecture Optimization Method

This section describes an abstract method for multi-objective design space optimization of software-intensive systems. This method has been synthesized

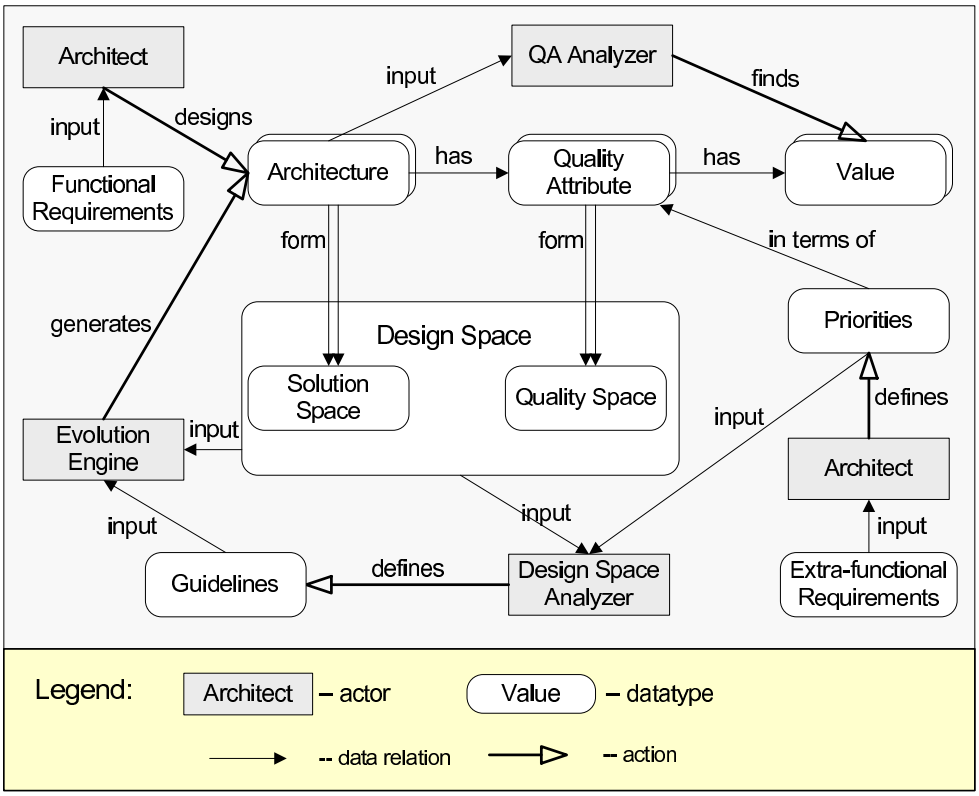

Figure 6.2: *Abstract method for multi-objective architecture optimization of software-intensive systems.*

from the above-mentioned approaches. The method specifies the process of designing and optimizing architectures with contradicting requirements on the multiple quality attributes.

Figure 6.2 depicts the abstract optimization method in terms of an iterative design workflow. The workflow is represented by logical blocks (actors and data types) and arrows (actions and data relations). The workflow diagram has two constraints: (a) only an actor can initiate a certain action and, (b) an actor cannot be a source of a data relation. The following paragraphs describe this workflow in four principal steps.

A. Definition of initial set of alternatives and evaluation of their QA values.
The input to the workflow is a set of functional requirements and constraints. In the design space identification phase, an architect creates an initial population of architectures based on the system requirements and his/her own

experience and intuition. A population of architectures forms a *solution space* [73]. Each of the architectures may be characterized by a set of quality attributes, e.g. task latencies, resource usage, performance sensitivity to load peaks, reliability, and cost. To obtain values of the quality attributes, the architecture is inspected by a specific *QA Analyzer* tool. This QA Analyzer determines (with some level of accuracy) the values of the relevant quality attributes that form a so-called *quality space* [73]. The available techniques that can be used for such a QA Analyzer, are outlined in Chapter 3.

The solution space and quality space together form a *design space* of the particular iteration. The design space can be represented by a multi-dimensional diagram in which the architectures are positioned along QA-related axes according to their predicted QA values.

B. Analysis of promising directions within the design space. The next step of the workflow iteration involves extending the architecture population. At this phase, the architect defines *priorities* between multiple quality attributes, based on the extra-functional requirements. The above-specified set of quality attributes allows to generate various kinds of trade-offs in which multiple performance properties can be involved[1]. Normally, one extra-functional requirement addresses one quality attribute. In case of conflicting requirements, the architect consults project stakeholders to identify the right priorities. As an example, QA priorities can be specified as weighted cost functions. The architect sets the priorities at the beginning of the optimization process and may tune the priority values during following iterations.

A more advanced way to prioritize the contradicting quality attributes is to employ *utility functions* [94]. In particular, utility functions allow dealing with the situations when a value of a quality attribute comes close to the required value, but still the corresponding requirement cannot be met. In case the utility function for this quality attribute is set to be low, the architecture can be accepted for further consideration.

These priorities (utility functions), together with the current design space (population of architectures including their quality attributes) are the input to a *Design Space Analyzer* tool. This tool plays a crucial role in the whole architecture optimization process. The main functions of the Design Space Analyzer are as follows.

1. *Evaluation of the current design space and finding the set of non-dominated architectures.* The evaluation process can be based either on the Pareto-frontiers methods [74], or on the various cost function approaches. In case of the Pareto-frontiers methods, the evaluation identifies the set of non-dominated architectures. If a cost function (e.g. weighted sums or

[1]Examples of performance properties forming the quality space are task latencies, processor, bus and memory usage, performance sensitivity.

products) are used, the result would be one specific order that reflects the realization of the different optimization objectives.

2. *Identification of weak points in the design of the non-dominated architectures.* This step requires some knowledge on how certain design decisions influence values of quality attributes. The initial data for extracting such knowledge can be obtained from the QA Analyzer output (e.g., task behaviour timelines, hardware resource load imposed by tasks). Based on this input, the Design Space Analyzer tool can identify weak points or bottlenecks in the design of non-dominated architectures. For example, a system has strict requirements on a response time of a certain task. The QA Analyzer tool finds/predicts the response time value by simulating the execution of the architecture. The Design Space Analyzer acquires the simulation timeline of the task and analyzes the bottlenecks in the system that led to the response time increase. Examples of such bottlenecks are: low buffer capacity or network bandwidth, high processor load due to other tasks, task blocking and deadlock.

3. *Identification of guidelines for generation of the next architecture populations.* Having the data about weak points and bottlenecks in the architecture designs and knowing how each particular bottleneck influences the architecture dependability, the Design Space Analyzer creates *guidelines* for generating a new set of architectures. The guidelines may specify: (a) the generation algorithm for the next iteration, (b) variable elements in the architecture (e.g. measures to improve quality), and (c) value ranges for these variable elements. For instance, the guidelines may specify for a system that processor frequency and buffer capacity could be reduced up to 50% of their current values without any consequences for high-priority QAs. At the same time, the guidelines may point out that the system bus bandwidth causes signal delays and should be increased. Although this example refers to hardware variations only, it may equally be possible to optimize in the software domain or in combinations.

In order to automate the analysis, the Design Space Analyzer may implement a machine-learning algorithm. This algorithm is supplied by an architect with some initial rules and examples for: (a) identification of weak points of an architecture leading to low values of corresponding quality attributes, and (b) construction of the guidelines for removal of these weak points, thus for generation of more balanced architectures.

C. Generation of new architecture alternatives.
The guidelines that are constructed by the Design Space Analyzer, together with the available design space, form an input for an *Evolution Engine*. The Evolution Engine is responsible for generation of new architecture population.

Possible working principles for the Evolution Engine are Monte Carlo Search, Simulated Annealing [76], Tabu Search [68], Genetic Algorithms [105, 51, 65] and Ant Colony-based Algorithms.

D. Evaluation of QA values of new architecture alternatives. The newly generated architecture population is sent to the QA Analyzer to determine their quality attribute values. Once the QA values are found, the architect may take a decision to stop the optimization process, or to continue with a next iteration. The criteria for the stop-decision are: (a) convergence point achieved - no enhancement seen in comparison to previous iterations; (b) requirements satisfaction - all relevant requirements are met; and (c) no time available for next iterations.

An important feature of this generalized iterative method is that the different techniques for QA analysis and architecture generation can be applied during the same process. We call this feature - *cascaded optimization*. Cascaded optimization allows varying the analysis and architecture generation techniques depending on the current situation. For instance, once the Design Space Analyzer notices that local search provided by the Simulated Annealing algorithm comes close to a local optimum, then it generates and sends a guideline to the Evolution Engine to change from local to global search algorithms (from Simulated Annealing to Genetic Algorithm) in order to explore a larger design space.

6.4 Mapping the Abstract Method on the DeepCompass Framework

The DeepCompass framework is fully compliant with the process of the aforementioned abstract optimization method. The DeepCompass process applies an iterative development (see Section 4.2) that includes the rapid construction of architectural alternatives, assessment of their quality attributes based on the architectural specifications and the trade-off analysis based on the Pareto-frontiers principle. Let us explain how the abstract method can be mapped onto the DeepCompass framework.

The initial set of architectural alternatives can be generated using the Repository and the Graphical Designer tools of the CARAT toolkit. These alternatives form the initial Solution Space. The functionality of the QA Analyzer is performed by the Preprocessor and Simulator tools, implementing our scenario-based method for quality attribute predictions (see Chapter 5). The predicted values of the quality attributes form the initial Quality Space.

The Solution and Quality Spaces define the Design Space, which can be explored with the Pareto-frontiers method, deployed in the DeepCompass framework (see Section 4.6). Pareto-frontiers help to identify the non-dominated

solutions and reason about the alternatives with the most balanced distribution of relevant quality attributes. The construction of the Guidelines for further optimization of the non-dominated architectures is not yet automated in the DeepCompass framework. An architect has to analyze how the design decisions influence the quality attributes for each alternative. Some typical issues for analysis points are the following: (a) which decisions lead to the system bottlenecks; (b) which factors impose low values of important quality attributes, (c) which decision influences multiple quality attributes and how.

Based on this analysis, an architect is able to construct the guidelines for further optimization of the non-dominated alternatives. These guidelines are formalized in terms of the rules for model transformations. Because this research is still ongoing, we only outline the possible types of model transformations in the following list.

1. *Change capacity of processor/memory/bus node.* Based on the identified bottlenecks in the hardware resource usage, an architect can adjust the deployment model by substituting the hardware nodes with different processor frequency (memory size and bus bandwidth).
2. *Change hardware topology.* This transformation enables changing the number of processing and memory nodes, as well as the topology of the communication infrastructure that links those nodes.
3. *Substitute software component.* A component can be substituted by another component, providing similar functionality, but possessing different characteristics of its quality attributes.
4. *Change deployment of software component.* A component can be redeployed from one processing node onto another one. If a component heavily uses an overloaded hardware node, it can be mapped on the less loaded node.
5. *Apply load balancer.* The same reasoning as in the previous bullet applies here, but with a different transformation. The load balancer is a software or hardware entity that distributes the input events onto two software components, providing identical functionality but deployed at different hardware nodes. The balancer may distribute events evenly, or based on the current loads of the corresponding processing nodes. The loads are monitored in real-time.

As a result, a new set of architectural alternatives is generated. These alternatives are the starting point for the next iteration of the optimization process. The optimization can be halted when the saturation point is achieved (no improvements are seen for last iterations), or once an architect obtains a solution that satisfies the requirements.

6.5 General Challenges in Architecture Optimization

In this section, we present the main challenges that an architect faces at the architecture optimization phase. The nature of these challenges is pluriform. For example, they deal with the sometimes huge design space, or the inherent inaccuracy of models and their parameters. Therefore, we only provide a brief list here.

Inability to find all Pareto-optimal solutions. The design spaces of most real-world applications are complex, if not infinite. As a result, a complete exploration of the design space is unfeasible. To perform a guided search, heuristics, like simulated annealing, genetic algorithms or Tabu search, are used in the abstract method. These heuristics help to search through complex design spaces. However, there is no guarantee that globally optimal solutions will be found. These heuristics often only produce suboptimal or locally optimal solutions. Nevertheless, these solutions are typically better than the original non-optimized architecture specification.

Inaccurate predictions of quality attributes. The outcomes of quality evaluations at an architectural level are always estimations of the real dependability attributes of the system in operation. The reason for this is that most parameters used for dependability evaluation, like the failure rate of a software component, cannot be quantified exactly at the design time. Furthermore, if off-the-shelf components are purchased, then the dependability evaluation relies on the correctness of the dependability parameters provided by the component vendor(s) [5]. Another source of inaccurate dependability evaluation results are the limitations of the dependability evaluation models and methods themselves. These models and methods make assumptions in order reduce the complexity of the dependability evaluation (see for an example in [36]).

Problems with formalizing and automating architecture transformations. Another requirement of fully automated architecture optimization approaches is the need for formal and automatic architecture transformations. This limits the selection of the architecture transformations to the class of dependability improving measures that can be formally specified and applied without intervention of the system architect. However, some transformations need additional information that must be provided by the system architect in order to apply the transformation. Examples are transformations that add extra components into the architecture. These components must be formally specified and basic component-based dependability metrics must be added to allow evaluations of the system dependability attributes after the transformation.

6.6 Conclusions and Future Work

Requirements on quality attributes such as performance, robustness and cost often conflict with each other, which make the development of dependable systems complicated. It is not always possible to design a system that fulfills all of its dependability requirements. Therefore, it is necessary to identify these conflicts early in the development process and to design an architecture with a balanced distribution of the quality attribute values. This balancing requires an iterative process, containing the phases of: (a) prediction of the quality attributes of the available architectural alternatives; (b) comparison of the alternatives with respect to multiple conflicting attributes; and (c) optimization of these alternatives to obtain a better balance between these attributes.

In this chapter, we have performed a comparative review of the approaches that address this architecture optimization problem. We have identified and outlined the techniques that these approaches deploy for each of the iterative phase of the optimization process. Our review suggests that none of these approaches consistently outperforms others in the quality of the derived solutions or the computational effort and that, in general, performance largely depends on the formulation of the problem and the nature of the potential design space.

Despite the wide diversity of the methods that we reviewed, significant commonalities were identified to enable us to define an abstract method for architecture dependability optimization. The abstract optimization method has an open and extensible character. For example, it was discussed that an architecture could be transformed with an evolutionary process, that could be even based on artificial intelligence. In this sense, the presented optimization method is not more than a skeleton. Such an abstract method can guide developers in the optimization of dependability-critical systems by providing a general reusable framework that will enable additional functionality (e.g. new heuristics) to be included, or specific actions to be taken to suit particular formulations of the optimization problem.

We have mapped the abstract method to the context of the DeepCompass framework. We validate parts of generic results of this chapter in two of the three case studies discussed in Chapter 9. As future work, we plan to complete the generalized DeepCompass method with several aspects. Firstly, we envisage the development of a self-learning engine that would enable automated identification of architectural problem points in the available alternatives, based on the predicted values of their quality attributes. The found problem points should be converted by the engine into the rules/guidelines for model transformation. Secondly, we need to formalize the specification of these rules, as well as the specification of possible transformations. Finally, we intend to develop an evolution engine for design space exploration, that would realize the automated transformations of corresponding models, based on the defined transformation rules.

Chapter 7

CARAT Software Toolkit

7.1 Introduction

Design and development of current software-intensive systems requires support from powerful Computer-Aided Software Engineering (CASE) tools. Such software tools, like Rational Rose [54] and ARTiSAN [4] , have proven to be efficient for various design phases of the software development. They provide a broad functionality from keeping the project documentation and checking the consistency between different diagrams, to state-chart simulation and code generation.

However, for the development of real-time embedded systems, the above software tools often lack a decent function for performance verification. Real-time systems are characterized by their strict end-to-end response-time, throughput, and robustness requirements. An early verification of these constraining requirements already at the design phase reduces technical risks and production costs. Software tools (like VTune [55] and HProf [91]) exist that provide full-fledged functionality for all kinds of performance verification and optimization, but they require source code of an application and cannot be used at the design phase, when the source code is not available.

The software tools, simultaneously providing facilities for design and performance-analysis, can be divided into two categories: commercial and academic tools. The LinuxLink [96] toolkit from TimeSys is an example of a commercial tooling environment which enables design, performance assessment and optimization for software products on Linux platforms. The performance assessment is provided by an embedded simulator based on Rate-Monotonic Analysis (RMA), which limits its applicability when a heterogeneous hardware platform

is used. Academic tools provide various analysis techniques, ranging from formal methods to simulation-based techniques. The Sesame environment [24] features modeling and simulation methods and tools for the efficient design of heterogeneous embedded multimedia systems. The Real-Time Calculus Toolbox [95] uses an efficient analytical approach with "min-plus" and "max-plus" algebra operators for resource load and event curves.

Presently, a trend is noticeable to build software for complex embedded systems according to the principles of component-based software engineering (CBSE). The amount of available CASE tools for component-based real-time systems is rather limited. The CB-SPE Tool [7] features graphical design and performance assessment of systems built from conventional software components. It adopts RT-UML profile annotations for components and composes these annotations into a system QN model at the component-assembly phase. Analysis of the QN model leads to the predicted system performance. The SEESCOA Tool [99] enables designing software systems from components with specified timing contracts and run-time monitoring of these contracts. Both of the tools provide solid performance-assessment functionality. However, these tools lack design-support features, like repository, large-scale visualization and automated code generation. The SAAM tool [59] based on the powerful SAAM method, provides architecture evaluation of various extra-functional properties. Unfortunately, the tool does not address performance analysis at sufficient level of detail and accuracy. The DARPA Evolutionary Design of Complex Software (EDCS) program [26] focuses on the development and integration of tools to support architecture composition and evaluation of CORBA-based systems. The Rapide tool is a primary example of EDCS. It allows to browse through and animate complex sequences of events generated by an architecture simulation. The tool is efficient for understanding and validating the complex behavior of distributed component-based architectures.

In this chapter, we present the CARAT toolkit for design and performance analysis of real-time component-based software systems. The toolkit guides an architect through the DeepCompass process providing functionality for the complete design cycle and consists of the following integrated tools: a repository, graphical editors for software and hardware architecture modeling, a model synthesizer, performance analyzer and visualizer.

The CARAT toolkit is different from the above tools in the sense that is fully dedicated to the analysis of real-time embedded systems that are constructed with software components and using CBSE principles. Moreover, the toolkit provides the functionality for a full iteration of the design cycle, including constructing architectural alternatives from individual components, analyzing their performance properties and evaluating trade-offs between multiple of those properties. The largest added value of the toolkit is that it automates the complex reconstruction of the system-wide behaviour from the models of independently developed components. The CARAT toolkit is freely available

for experiments from [10].

This chapter is structured as follows. Section 7.2 describes the architecture of the toolkit. Section 7.3 discusses the benefits and limitations of the toolkit. Section 7.4 concludes the chapter.

7.2 Architecture of the CARAT Toolkit

The CARAT toolkit is implemented in Java and realized as a number of Eclipse plug-ins. This enables easy installation and high portability on any well-known platform. The data specification and exchange between these plug-ins are organized by XML-based model structures. The architecture of the toolkit, depicted in Fig. 7.1, consists of the following components: Repository, Graphical Designer, Model Preprocessor, Performance Analyzer, Visualizer, Statistics Reporter and Code Generator. A brief description of these components is given in the following paragraph.

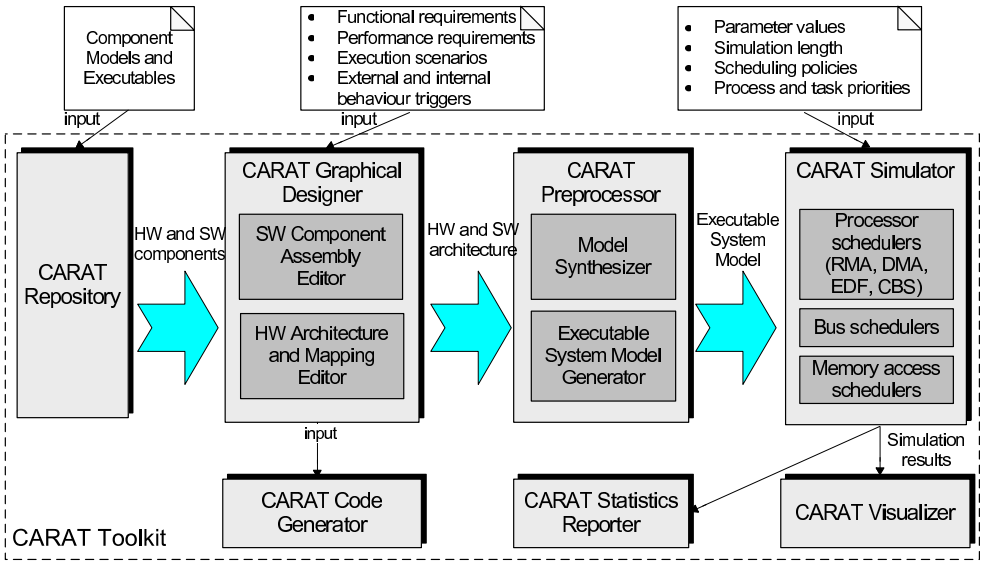

Figure 7.1: *Architecture of the CARAT toolkit.*

- The *CARAT Repository* provides storage and retrieval of executables and various models of both software components and hardware IP blocks.
- The *Graphical Designer* contains two editors for constructing (a) service instance assemblies, and (b) hardware resource topologies with assigned deployment (mapping) of the service instances.

- The *Preprocessor* takes the data from the Repository (model specifications) and from the Graphical Designer (defined architectures), and synthesizes the models into an executable system model.
- The *CARAT Performance Analyzer* uses this model as an input for virtual scheduling of the tasks on the corresponding hardware resources. The Scheduler outputs an execution timeline for each task for every hardware resource (processor, memory bank and bus).
- The *CARAT Visualizer* interactively draws the predicted timelines.
- The *Statistics Reporter* provides the designer with a broad range of data on the obtained performance properties.
- The *Code Generator* constructs the application "glue-code". The glue-code instantiates and binds the software service instances according to the specified assembly and prepares the deployment of the instances on the defined hardware nodes.

The remainder of Section 7.2 presents the functionality and interdependencies of these CARAT components in detail.

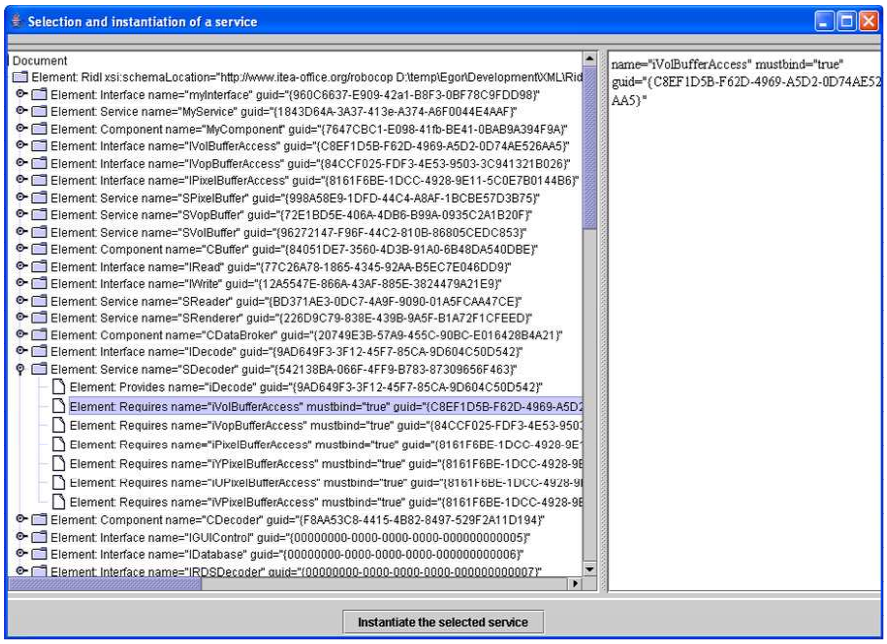

Figure 7.2: *Example of access window to the CARAT Repository.*

7.2.1 Repository

The Repository enables remote storage and retrieval of third-party component executables and their corresponding models. It allows a designer to search for services satisfying input criteria (functional and extra-functional requirements) and to view the service specifications. The Repository (see Fig. 7.2) is accessible from the Graphical Designer, hence, service instantiation by positioning a service on the editor plane.

7.2.2 Graphical Designer

The Graphical Designer (see Fig. 7.3) provides graphical means for modeling of a system architecture. The tool contains two complementary window views, where an architect can interactively construct the architectural models (see Fig. 7.3). The bottom window view is the SW Assembly Editor, in which the scenario models are constructed. The top window view represents the HW Architecture and Mapping Editor, in which the deployment models are specified.

In the SW Assembly Editor, a designer instantiates selected services and binds the provides and requires interfaces, thereby defining the static structure and communication topology of a software system. For the services whose behaviour and resource usage depend on parameters, a designer should define the parameter values. Besides this, a designer may also specify triggers (environmental/user events or interrupts) that stimulate activities in the service assembly. The set of triggers together with the parameter values and service assembly form a scenario model (defined in Section 5.4.2).

Fig. 7.3 depicts a snapshot of the design process of a Car Radio Navigation (CRN) system, for which we have validated our DeepCompass framework and the CARAT toolkit (see Chapter 9).

In the SW Assembly Editor, we instantiate the following ROBOCOP services in order to satisfy functional requirements: the Man-Machine Interface (MMI) service, the Navigation service and the Radio service. The MMI service instance provides its functionality via IGUIControl interface and requires to be bound to IParameters and IDatabase interfaces. The Navigation component provides IDatabase, IRDSDecoder interfaces and requires IGUIControl interface. The Radio service instance provides IParameters, IRDSReceiver interfaces and requires IRDSDecoder interface. By binding the interfaces of the same type, we define possible communication between the service instances.

These three service instances are not active (i.e. they have no active processes implemented inside the components). The system behaviour can be triggered by the user or environmental events. To emulate these events, we create and parameterize three stimuli (dark-grey boxes in the SW Assembly Editor) that actually trigger the behaviour of the CRN system. These stim-

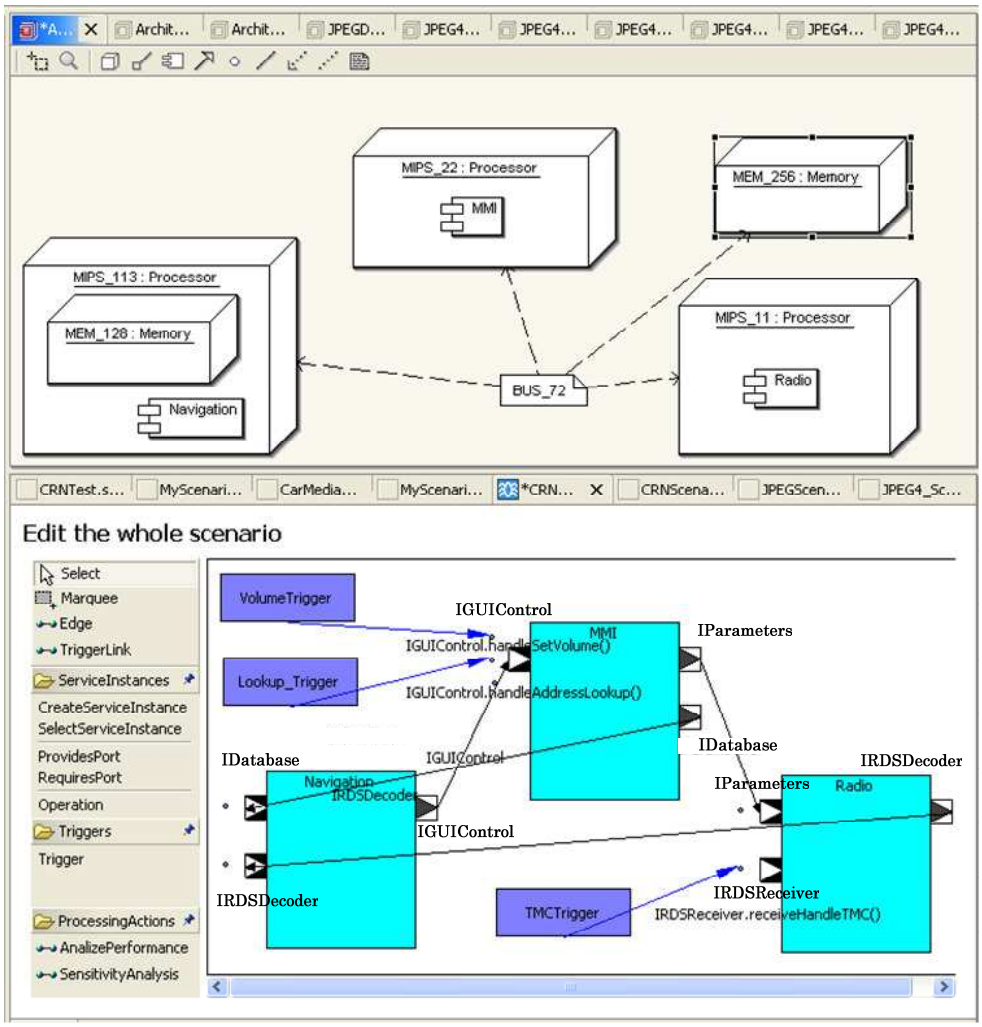

Figure 7.3: *Graphical Designer consisting of HW Architecture Editor and SW Assembly Editor, showing the CRN example.*

uli include (a) the VolumeTrigger emulating the user event "change the sound volume", (b) the LookupTrigger emulating the user event "find and retrieve an address" and (c) the TMCTrigger emulating the Traffic Message Channel (TMC) messages arrival. We connect these triggers to the service operations that they activate, once their events occur.

These three events may occur in parallel. Therefore, we set all three triggers to form a critical scenario. This scenario is critical because it may impose a resource overload or create a hazard for fulfilling the performance requirements.

Once the scenarios are defined, the hardware architecture and deployment of service instances can be specified in the HW Architecture Editor (the graphical panel located at the top in Fig. 7.3). The editor allows (a) selection of the processors, memory blocks and communication lines from the repository, (b) creation of an arbitrary topology from the selected hardware blocks, and (c) mapping of the involved service instances onto the hardware blocks. The processors and memory blocks can be connected by communication lines in arbitrary styles, like star, token ring and mixed topologies. A memory block can be specified as a local (in processor) or global memory. Conventional services can be mapped onto processing nodes, while virtual services representing memory buffers can be mapped only onto memory blocks.

7.2.3 Preprocessor and Performance Analyzer

The *CARAT Preprocessor* tool is a computationally complex part of the toolkit. The tool performs the algorithms for synthesis of the executable system model as shown in Section 5.5. As an input, the tool takes the scenario and deployment models, as well as the models of the involved software components and hardware nodes. These models are synthesized into the executable system model specifying tasks running in the system (see Section 5.4.4).

The *CARAT Performance Analyzer* provides a set of schedulers for processors and communication lines and enables simulation of the executable system model. The generic algorithm for a simulation scheduler is given in Section 5.6.1. A designer enters the configuration details as an input for the simulation (see Fig. 7.4). These details include simulation period, simulation type (worst, best or average case), types of scheduling algorithms for hardware resources and process/task priorities if needed. The simulation time-unit can be set to millisecond or microsecond scale.

With the configuration settings defined, the executable system model is simulated resulting in (a) the task-execution timeline for each processing node and (b) the timelines of data occupancy for busses and buffers. Besides this, the results include the identified maximum latencies and resource loads for tasks, data throughput and total utilization of hardware resources.

The detailed presentation of the simulation results is given in the next subsection describing the Visualizer and Statistics Reporter tools.

7.2.4 Visualizer and Statistics Reporter

The CARAT Visualizer shows two types of resulting data: a synthesized message-sequence chart (MSC) for every task, and the task-execution timelines of the involved hardware resources.

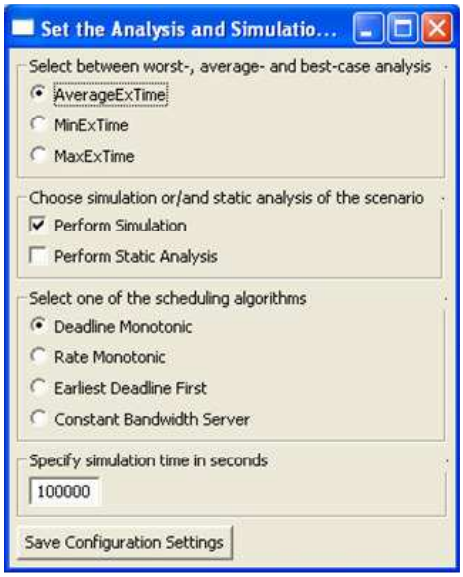

Figure 7.4: *Configuration window for simulation.*

For the above-specified CRN system scenario, the CARAT Visualizer has depicted three MSC diagrams (see Fig. 7.5). For example, the second MSC diagram shows that the task instance is triggered by TMCTrigger every 3,000 ms. The first triggered operation is receiveHandleTMC() of the Radio service. This operation calls the decodeTMC() operation of the Navigation service, which, in its turn, invokes the MMI.updateScreen() operation. The updateScreen() operation is a leaf operation in the call-graph tree. The task instance completes its execution when all return calls are made.

The predicted task-execution timelines for the three processors and the bus load in the CRN system are given in Fig. 7.6.

The diagram should be read as follows. The timeline of the processor MIPS_22 shows that the three tasks share this resource and interleave with each other. For each task instance, the timeline shows the start, execution and completion times together with the task deadline. Fig. 7.6 shows that the first instance of the TMCTrigger_Radio task fails to meet the deadline. The analysis of the available timelines results in the conclusion that this happens due to the high MIPS_22 processor occupancy which is caused by the higher-priority task VolumeTrigger_MMI.

The bus-load timeline shows the available bus bandwidth at each moment. The numbers on top of the bus-usage peaks specify IDs of the tasks transferring data over the bus at that moment.

The *Statistics Reporter* tool gathers all relevant information from the simu-

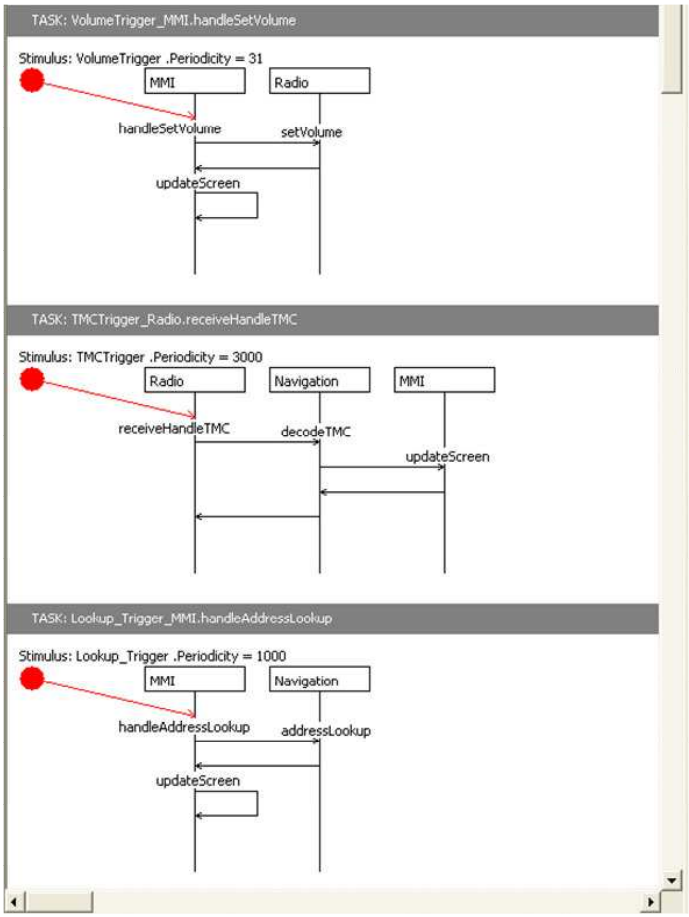

Figure 7.5: *Synthesized message-sequence charts for the three tasks in the CRN system.*

lation, processes it and stores the data in a text file. The performance statistics resulting from the simulation of the CRN system is depicted in Fig. 7.7. The analysis of the CRN system shows that the task initiated by VolumeTrigger takes 90.04% of the MIPS_22 processor. The total load (by all three tasks) of this processor equals to 93.76%.

7.2.5 Code Generator

This tool takes the scenario and deployment models as an input and generates C/C++ application glue code. This glue code implements loading and instantiating of services, as well as binding their provides and requires interfaces. The generated code also includes the memory-release functionality upon

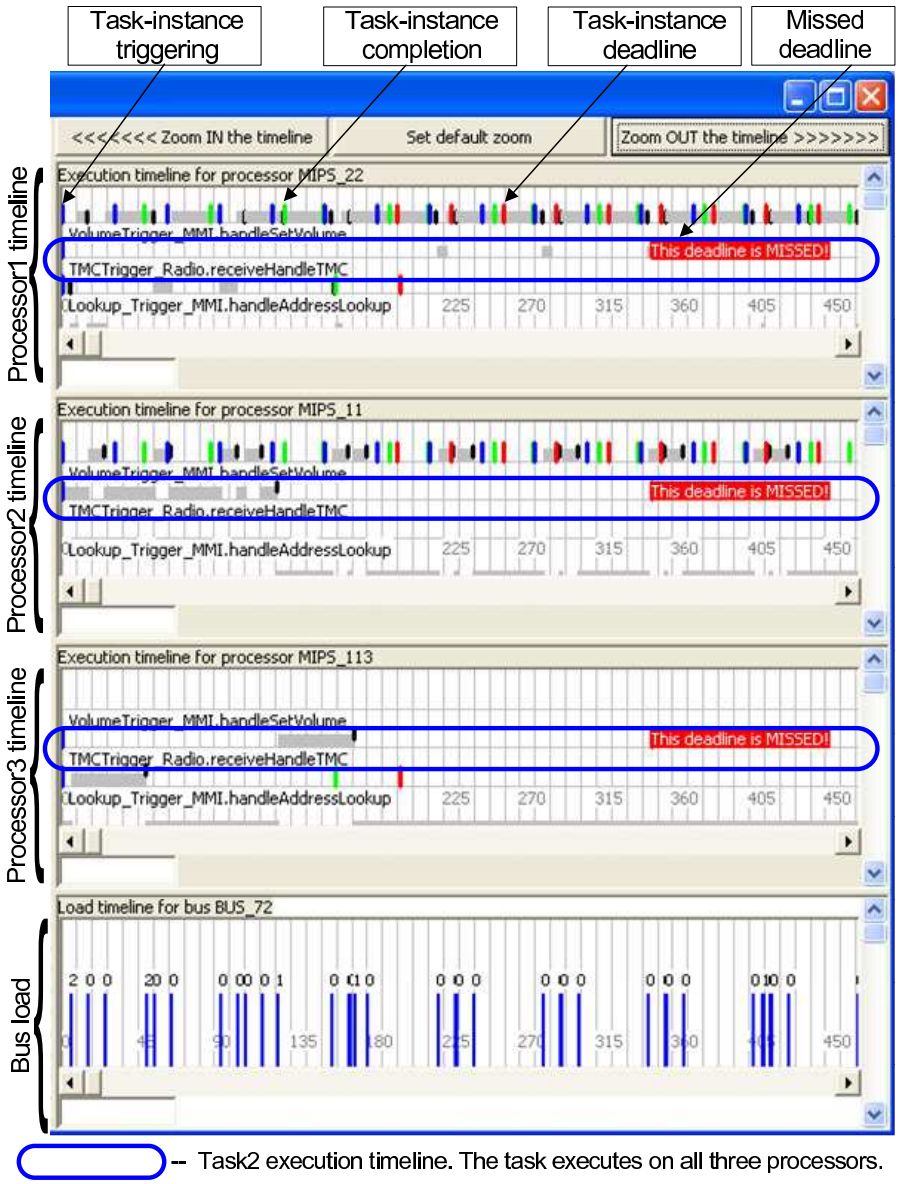

Figure 7.6: *Predicted task-execution timelines for three processors and bus-load timeline.*

application completion. The outcome of the toolkit is a complete executable code of the application and individual components.

```
******************  CPU MIPS_22 utilization statistics **********

CPU util of task VolumeTrigger_MMI.handleSetVolume  =  90.04%
CPU util of task TMCTrigger_Radio.receiveHandleTMC  =  0.92%
CPU util of task Lookup_Trigger_MMI.handleAddressLookup  =  2.8%
Total CPU utilization  =  93.76%


******************  CPU MIPS_11 utilization statistics **********

CPU util of task VolumeTrigger_MMI.handleSetVolume  =  28.98%
CPU util of task TMCTrigger_Radio.receiveHandleTMC  =  3.64%
CPU util of task Lookup_Trigger_MMI.handleAddressLookup  =  0.0%
Total CPU utilization  =  32.62%


******************  CPU MIPS_113 utilization statistics *********

CPU util of task VolumeTrigger_MMI.handleSetVolume  =  0.0%
CPU util of task TMCTrigger_Radio.receiveHandleTMC  =  1.76%
CPU util of task Lookup_Trigger_MMI.handleAddressLookup  =  4.4%
Total CPU utilization  =  6.16%

Task VolumeTrigger_MMI.handleSetVolume  was executed  3235 time(s)
Number of missed deadlines  =  0
Maximum latency             =  0
Best completion time        =  49
Worst completion time       =  69
Task TMCTrigger_Radio.receiveHandleTMC  was executed  48 time(s)
Number of missed deadlines  =  48
Maximum latency             =  201
Best completion time        =  414
Worst completion time       =  551
Task Lookup_Trigger_MMI.handleAddressLookup  was executed 105 time
Number of missed deadlines  =  2
Maximum latency             =  109
Best completion time        =  161
Worst completion time       =  310
```

Figure 7.7: *Example statistics on performance properties.*

7.3 CARAT Toolkit Properties

In this section, we briefly discuss CARAT properties important for any software tools, such as completeness, robustness and efficiency.

Completeness and Consistency Checks. The CARAT toolkit provides completeness and consistency checks between various design diagrams. It identifies (a) missing component models in the repository, (b) erroneous bindings between provides and requires interfaces, (c) absence of service instances on the corresponding deployment diagram, and (d) missing hardware connections for defined interface bindings.

Robustness and Efficiency. We have validated the toolkit by designing and deploying a number of embedded systems (see Chapter 9). Let us provide a few details on the simulation speed of the CARAT Simulator. For a relatively simple system (three tasks running on three processors for 10 million time-units), the simulation completes within 2-10 minutes. Our experiments have shown that the simulation time has a close-to-linear dependency with each of the following system properties: (a) the number of tasks, (b) the number of operations involved, (c) the number of software components, and (d) the number of hardware units. If multiple properties grow at the same time, the required simulation time will grow exponentially.

7.4 Conclusion

We have presented the CARAT toolkit for design and performance analysis of component-based systems, which are deployed on heterogeneous multi-processor platforms. The toolkit supports the complete design cycle of the DeepCompass framework starting from component selection from the repository and ending with a code generation for an application. The toolkit enables performance analysis at the early design phase, when the source code is not available, and automates the complex reconstruction of the system-wide behaviour from the models of independently developed components.

The benefits of the toolkit occur in multiple ways. Firstly, the toolkit enables performance predictions for heterogeneous multi-processor platforms by (a) provision of a wide set of protocols and virtual schedulers for simulation, (b) supporting active and passive components, and (c) multi-processor specification of component resource requirements. Secondly, the framework and its toolkit are generic with respect to various application domains and architectural styles. For instance, the toolkit can be applied to systems designed in "pipes-and-filters", "blackboard" or "client-server" architectural styles. Thirdly and finally, the toolkit enables smooth transitions through design phases by keeping all data at a single location and by providing seamless communication between its modules.

Chapter 8

Survey on Scenario-Based Performance Analysis

8.1 Introduction

As the complexity of software-intensive systems is constantly growing, the analysis of their behaviour and quality attributes at the early architectural phases becomes a non-trivial task. They often provide hundreds of functions to the end-user and react on dozens of events from the environment. The software itself implements multiple processes, running in parallel and featuring complex behaviour with interleaving execution paths or traces.

From the other side, in order to carry out complete and thorough performance analysis of such a system, an architect should explore all possible behavioural patterns, system configurations and execution paths. However, due to the strict time-to-market and cost requirements for industrial products, an architect is always limited in terms of time and resources. In these circumstances, *scenarios* can help to perform architecture evaluation within short time limits and with minimal efforts. Scenario-based analysis of software architecture is increasingly seen as an elegant, cost-effective means of controlling the risks inherently occurring in architectural design. Usage of scenarios allows architects to focus only on the subset of important (common, critical) execution paths and configurations, in which the system may fail in satisfying its performance requirements.

A scenario can be defined in a number of ways. In the general case, a scenario is considered as a brief description of some anticipated or desired use

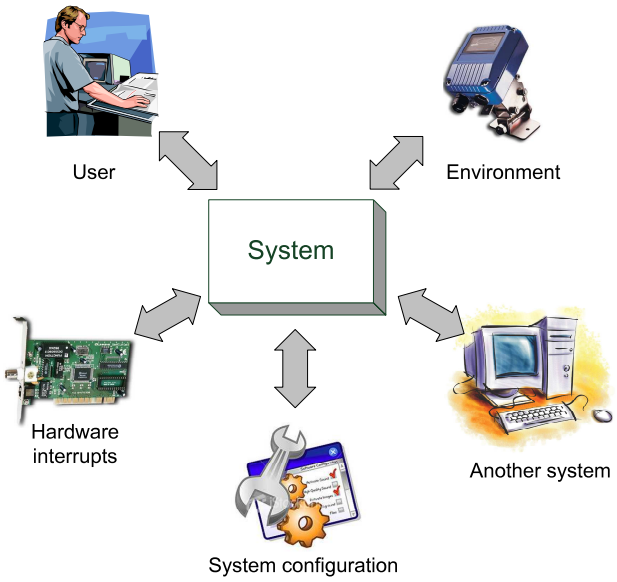

Figure 8.1: *Factors triggering scenario execution in a system.*

of a system [1]. Extending this definition, a *scenario* is defined as a set or sequence of interactions between a system and its environment (users, sensors, other cooperating systems), which lead to specific execution configurations of a system. For instance, a digital TV set-top-box (STB) which decodes a standard-definition video stream is one scenario, while the same STB decoding a high-definition video stream will be another scenario. In our work, we consider both types of scenarios in our performance prediction method.

In the last decade, scenario-based architecture evaluation methods received serious attention both from the industry and academia. This is motivated by the potential advantages that scenarios bring to system analysts and architects: (a) enable deeper understanding of the system behaviour, performance bottlenecks and usability, (b) allow better decomposition of the functional blocks of the system, and (c) reduce time and cost for analysis of functional and extra-functional system properties.

Initially, at the late 1980s, scenario-based analysis was adopted in the community of usability experts [53]. Scenarios proved to be effective in specifying user actions and behaviour. Identification of usage scenarios (use-cases) became an integral part of the requirements engineering phase within the development processes. Later on, architects started using scenarios for assessment of modifiability, evolvability and maintainability of system architectures. For this purpose, a number of mature scenario-based methods, such as SAAM [60]

and ATAM [61] have been developed in academia and adopted later in the industry. Within these methods, a scenario is defined as a set of anticipated post-development changes in the system requirements. For every scenario of this type, a system architecture can be evaluated with respect to modifiability or maintainability.

Recently, scenario-based approaches have attracted attention of architects and researchers working in the field of performance-critical systems. For such systems, performance and behaviour-related quality attributes are the most important properties to evaluate at the architecting phases. In this field, scenarios help to clearly define and limit the scope of time-critical tasks for further performance evaluation. Normally, such a scenario represents a set of concurrent tasks running at a specific system configuration. The set may contain not only time-critical tasks, but also tasks without real-time requirements. Once the scenario is executed, the behaviour of time-critical tasks and the total usage of hardware resources can be analyzed.

OMG introduced scenarios as a standard in its UML Profile for Schedulability, Performance, and Time [40]. Scenario-based approaches for performance evaluation have been proposed for various application domains, i.e. object-oriented software systems [64], component-based software systems [79], mobile networks [81] and navigation systems [84].

Despite the benefits that scenario-based approaches bring to practitioners, there are a number of challenges that an architect faces once deploying scenario-based evaluation of architecture quality attributes. Table. 8.1 describes these challenges in detail.

Due to the fact that our performance prediction technique is scenario-based, we need to properly address these challenges and provide recommendations and guidelines for scenario identification and selection. We have conducted an empirical case-study survey by interviewing leading industrial architects about their experience and methods used for scenario-based assessment.

The case-study survey consists of written questionaires and results of face-to-face interviews. In the interviews we have asked architects to share their experience in applying scenario-based performance analysis and their ways to solve the challenges specified in Table. 8.1.

This chapter presents the case study description and obtained results. It will be shown that the survey reveals interesting findings. For example, we have found that the rate of the scenario-based approaches among these techniques is quite high. The architects explained that full-search or exhaustive analysis techniques are the main alternative to scenario-based analysis.

The survey identified main benefits and drawbacks of scenario-based analysis in comparison to full-search analysis. All the interviewees mentioned that scenarios are inevitable for the analysis of large and complex systems, because they allow to focus only on relevant architectural issues. This saves time and resources while exhaustive analysis is hardly applicable in this context. Besides

Table 8.1: *Challenges in scenario-based analysis.*

Challenge Type	Description
Completeness of analysis	Scenarios do not guarantee complete architecture exploration, because they cover only a subset of all possible system configurations, external stimuli and behaviour traces
Scenario identification	There are no straightforward ways to identify all relevant and representative scenarios for analysis. The analyst can only rely on the expertise of the stakeholders and its own experience.
Scenario selection	A complex system may be represented by thousands of relevant scenarios. Due to the time and resource limitations these scenarios should be scoped to the set of most critical ones. Currently, there are no guidelines available for accurate critical scenario selection.
System-wide interpolation of the obtained results	In order to draw conclusions on the quality attributes of a global architecture, the results obtained from analysis of individual scenarios should be interpolated for the whole system. At present, architects lack well-established techniques for such interpolation.

this, scenario-based analysis is very efficient for product families and evolving architectures, and enables only local analysis for small changes in architecture and requirements. However, scenario-based approaches showed to have the following drawbacks: (a) relatively low prediction accuracy due to incompleteness of the analysis, (b) challenge in identification and selection of all relevant scenarios for consideration, (c) technical risk of failing to meet requirements after one or more scenarios were missed.

The chapter provides guidelines given by architects for both the scenario identification and selection processes. Moreover, it exposes a number of risk management methods and recommendations on the type of systems for which scenario-based analysis can be applied.

The chapter is structured as follows. Section 8.2 describes the design of the case study. In Section 8.3 we present the findings of the case study and summarize various recommendations for deploying scenario-based analysis. Section 8.4 provides a comparison between scenario-based and full-search types of analysis and justifies the feasibility of applying scenarios for assessment of large industrial systems. Section 8.5 concludes the chapter.

8.2 Design of the Case Study

8.2.1 Data Collection Procedures

Six system architects with profound background in architecture design and evaluation have participated in our study. They were invited based on their knowledge in the real-time and embedded systems domain. The interviewees were working as architects in middle- and large-scale projects in international companies and institutes, such as Philips, Nokia, ESI, ASML, ICS, Sober-IT, LogicaCMG and HUT.

The data collection consisted of written questionnaires and follow-up face-to-face interviews. The interviews have been recorded and informally analyzed for patterns in responses. Moreover, we asked the interviewees to be critical and objective in their answers.

8.2.2 Issues Investigated

This section briefly describes the issues in the scenario-based analysis that we have investigated. Though we are mainly focused on the approaches for accurate identification of critical and relevant scenarios for architecture development and performance assessment, the case-study survey also addressed related topics like requirements analysis, architecture design methods and architecture evaluation techniques. It started with gathering data on the background knowledge and experience of the architects. This data includes: years of experience, number and size of the projects participated in the past, and the application domains of the products designed by the architects.

Performance Requirements Realization and Architecture Evaluation. This part of the interview was focused on obtaining the general view of the architects on the architectural process, and more specifically, on the requirements analysis and architecture evaluation (see Fig. 8.2). Here, we collected empirical data on the architect's methods for addressing (realizing) performance requirements in the architectures. In detail, we planned to obtain the interviewee's experience on the following aspects: (a) clearness of the performance requirements description, (b) methods for the requirements realization in architectures, (c) most challenging performance requirements for realization, and (d) deviation between the values specified in requirements and the measured (real) performance values.

This section discovers challenges and problems that architects experience during requirements analysis and architecture evaluation, as well as opens up solutions that the architects use to address the challenges.

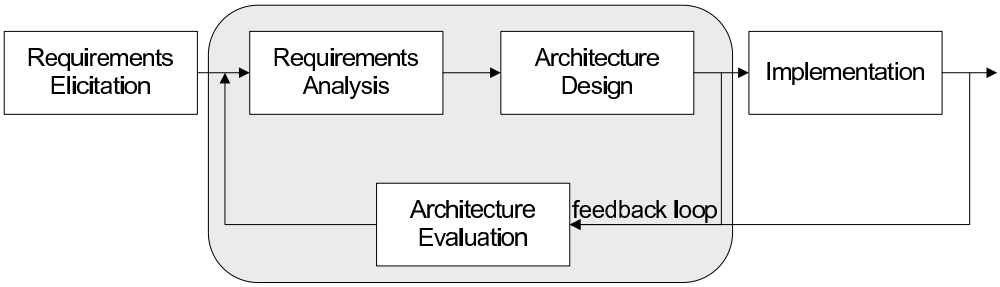

Figure 8.2: *Generic development process with highlighted architectural phases.*

Scenario Identification and Analysis. This section of the interview focused on collecting the architect's experiences, methods and techniques used in deploying scenario-based approaches for performance analysis. The following main topics are addressed in this section.

- The usage frequency of scenarios employed for the architecture design and assessment phases.
- Approaches for identification of relevant (critical, important) scenarios for performance analysis.
- Criteria for selection of only relevant scenarios from all possibilities.
- Common difficulties in the accurate identification of the performance-critical scenarios.
- Average rate of missing critical scenarios and the impact of that on the further project development.

The concluding part of the interview involved discussion on the feasibility of scenario-based approaches for reliable and accurate performance assessment. In this part, we asked architects to justify applicability of scenarios for real-time system design and assessment.

8.3 Main Findings of the Case Study

The findings of the case-study survey are summarized in this section and are organized by the issues investigated. Besides that, we derive recommendations for the usage of scenarios that emerge from these findings.

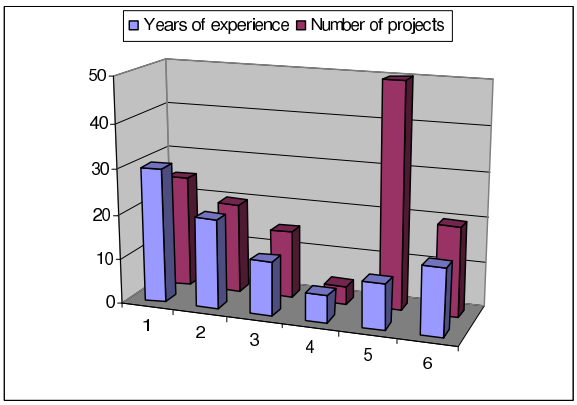

Figure 8.3: *Architect's profile: years of experience and number of projects.*

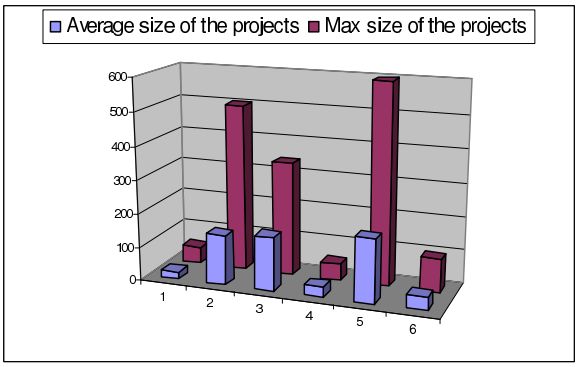

Figure 8.4: *Architect's profile: average and maximum size of the projects.*

8.3.1 Background of Interviewees

The interviewee's experience and background data is summarized in Fig. 8.3 and Fig. 8.4. The architecting experience varies from 6 to 30 years, on the average about 15 years. The number of projects, in which they have been involved as architects differs from 5 to 50, with the average of 22 projects. In total, the architects have reported their experience on 134 projects. The size of these projects varied from 1 to 600 manyears, averaging to 273 manyears.

The scope of the application and knowledge domains of the interviewees was quite broad. In Table. 8.2, we present these domains according to non-orthogonal criteria.

The above-mentioned data shows that our selection of the highly-experienced candidates is fairly broad and allows gathering of a wide spectrum of data.

Table 8.2: *Reported experience in application domains.*

Criteria	Application Domain
Real-time domains	Soft-, firm- and hard-real-time systems.
Hardware platforms	PC-based, embedded systems, systems-on-chip, MP-SoC, distributed systems.
Application domains	Medical, navigation, surveillance, measurement, logistics systems, robotics, mobile phones, wafer-steppers, set-top-boxes, 3G networks, switching, base stations, DTV and HDTV systems.

8.3.2 Performance Requirements Realization and Architecture Evaluation

A. Clearness of performance requirements

Most of the architects reported insufficient accuracy and clarity in the performance requirements specifications (see Table 8.3.2). The general opinion was that the stakeholders/customers primarily focus on functional requirements and expect the performance will be delivered by a product by default. In most cases, customers put attention to the performance requirements, once the product starts experiencing problems at the testing phases. These facts impose high tolerance to vague, unclear and imprecise performance requirements from the customer side.

Table 8.3: *Clearness of performance requirements.*

Interviewee ID	**1**	**2**	**3**	**4**	**5**	**6**
Were the performance requirements stated clearly in your projects?						
Scale from 0 (not) to 5 (very clear)	1	2-4	4	3	1-5	1

Another consideration, mentioned by the architects was that the performance requirements are often specified in the problem-domain terms (the decoder shall provide high-quality image, the user should not notice skipped frames, etc) but not in the technical quantitative measure (milliseconds, bits per second, etc.). Therefore, the technical analysis of requirements and their conversion to the values within the technical domain is always needed.

Citation: In order to be successful, an architect should understand the application domain better than his customer/stakeholder

B. Challenges in Realizing Performance Requirements
The architects encounter the following generic list of challenges.

- *Vagueness of requirements.* Conversion from application domain-specific requirements to the technical domain-specific requirements demands deep understanding of the application domain.
- *Hardware resources constraints.* In most cases, the hardware resources are predefined, and some real-time requirements are hard to realize for the given hardware platform. Moreover, this reduces the freedom in selecting the most suitable solutions.
- *Tracing requirements through architecture modules.* Such type of performance requirements as latency is a property of execution trace (task) in a system. The execution trace may pass through many architectural modules and can make complex loops. For complex systems, the identification of the trace route and the modules involved is not trivial.
- *Dependencies between real-time tasks.* The real-time tasks may interleave and block each other in complex patterns. Therefore, for realization of the related real-time requirements, the global analysis of the execution architecture is needed.
- *Hidden influence from other entities.* An architect should be able to anticipate indirect influence on the performance properties from other architectural entities, such as, hardware interrupts, cache misses, bus arbiters, memory access and hardware mismatches. Besides this, a deployed technology may bring some indirect influence on the system performance (e.g. Java garbage collector).

The above-mentioned items show the diversity of the challenges the architects face at the design phases. The following findings on the case study describe the consequences of the challenges experienced by the architects.

C. Realized-against-required deviations of performance
The architects encountered that every project, in which performance requirements were specified, experienced deviations in the measured performance properties (see Table 8.4). In some cases, these deviations were quite substantial (factor of 10) and caused a complete redesign of a system. Those deviations resulted in failure of product deadlines and increase of the total development cost.

The architect's experience proved that the performance properties are difficult to address in an architecture in an accurate and predictable way. The

Table 8.4: *Measured-from-required deviations of performance.*

Interviewee ID	**1**	**2**	**3**	**4**	**5**	**6**
How often did the real (measured) performance deviate from the required (planned) performance?						
In percentage	20-40	80-100	NA	20-40	40-60	60-80
What are the worst performance deviations you have experienced?						
In percentage	>500	1000	400	25	1000	25

Facts: The following reasons have caused the worst deviations: (a) Float operations has been used for integer-based processor in MR-Scanner, causing latency to be 10 times longer than required. (b) A mismatch between bus arbiter, DMA (Direct Memory Access) and network topology in HDTV (High Definition TV) system caused latency to be 4 times longer than allowed. (c) Introduction of design-patterns for object-oriented software caused substantial slow down (10 times) of navigation system operations.

following topics encounter what types of performance properties are the most difficult to deal with.

D. Types of performance properties most difficult to realize

In this part of the interview, we explored which types of performance requirements cause major problems during architecture design and assessment. In other words, what properties are the most difficult to realize in an architecture and to assess (predict) at the architecture evaluation. Table. 8.5 presents the architect's experience and opinions on this issue.

The answers show commonalities in the sense that requirements related to latency and throughput are the most challenging to realize. This conclusion is exacerbated by the fact that in many hard real-time systems, both of the requirement types are the most critical ones.

E. Approaches for realization of performance requirements

Once the architects determined the challenges in addressing and evaluating the performance requirements in architectures, we have tried to elicit their approaches in facing these challenges in a reliable and predictable way. We have gathered a wide range of answers that are reported in this subsection and structured by the following types of architectural phases: (a) requirements analysis,

Table 8.5: *Most challenging performance properties to realize.*

	What type of performance properties showed to be most difficult to address and why?
1	**Throughput** is most difficult to predict and realize, because it is system-wide property. To predict throughput values, all execution processes, hardware topology and synchronization issues should be jointly considered.
2	**Latency, throughput, processor and memory usage**. One of the main reason for the difficulties is the unpredictable and uncontrollable behaviour of conventional operating systems. For instance, the task scheduling-algorithms of Linux and Windows feature a "fair-use" protocol. In this case, an OS decides itself which concurrent task to execute.
3	**Latency** requirements become hard to realize and verify in case of heavy memory fragmentation, task interleaving/blocking and cache misses. Besides this, latency depends on the total CPU load on the system, e.g. 20% increase in CPU load may impose a 200% increase in critical task latency.
4	**Latency** is a property of an execution task. An execution task comprises many functions located at different architectural layers (deep layer propagation). Moreover, a task interleaves with other concurrent tasks and may be blocked by them. The analysis of task interleaving and blocking patterns as well as tracing the task functions through layers requires detailed and complex performance evaluation methods and tools.

(b) architecture design, and (c) architecture evaluation. The approaches are depicted in Fig. 8.5.

Note that the approaches are orthogonal and can be combined, e.g. iterative development and modeling often go along. The approaches are characterized by the architectural phases they are applied to, and by the number of references (weight) given by the interviewees (from 1 to 6).

- *Iterative evaluation. Phases: All. Mentioned: 4 times.* Initially, when no architecture and hardware modules are available, the architects evaluate alternatives and performance at the coarse grain level. After developing hardware platform , the architects prototype the most critical software entities and check for possible bottlenecks in the system. When the SW/HW architectures are available, the architects assess the performance at a fine grain level and optimize the global system architecture. At the testing phase, the architects monitor the performance, validate it against the requirements and optimize architectural elements in detail if needed.

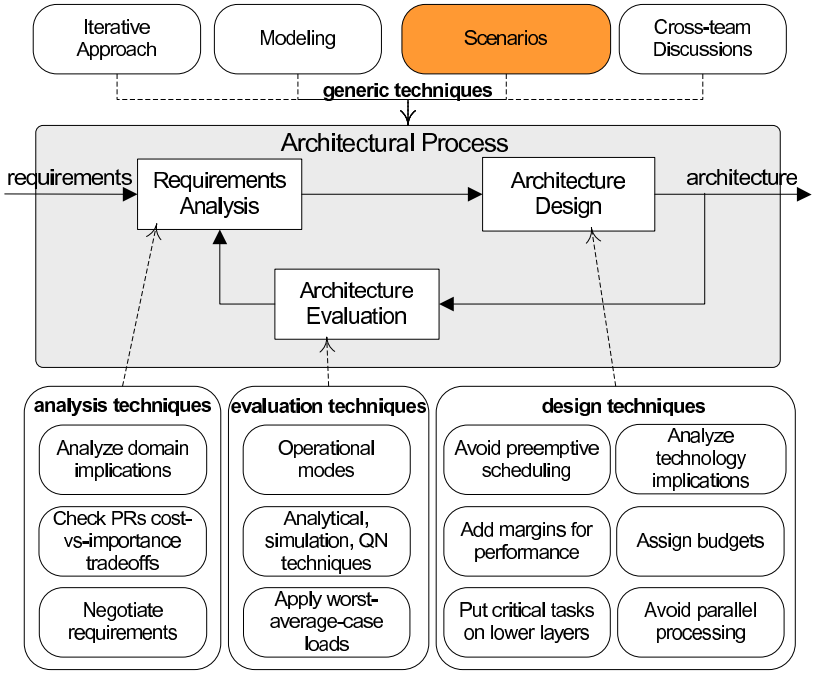

Figure 8.5: *Approaches for performance requirements realization.*

- *Scenarios. Phases: All. Mentioned: 4 times.* The architects extract the common or important usage scenarios from the requirements and convert them into system execution scenarios. Then, they evaluate the architecture by analyzing the execution scenarios and by looking at the propagation of the scenario traces through the architectural blocks. Finally, architects validate the performance of the system against the requirements by simulating (monitoring) the execution of the scenarios.

- *Modeling. Phases: All. Mentioned: 3 times.* Modeling was considered a good way to represent complex entities at the high level of abstraction and suitable for removing irrelevant details. Models were used to specify different levels of abstraction, thereby helping to understand complex structures and behaviour in a nested manner. Models were used in two way: (a) to simplify the representation of already built entities, and (b) to specify (in a structured and clear way) how the entities should be constructed.

- *Cross-domain discussions. Phases: All. Mentioned: 3 times.* The interviewees stressed that development of current systems involves signifi-

cant cross-domain knowledge. For instance, medical systems incorporate knowledge on imaging, electronics, physics and medicine. Therefore, an architect is to be able to talk to and understand experts from different domains. Also, high-quality cross-team communication was found crucial, i.e. an architect should ask the right questions and give clear answers to people from the requirements, implementation, test and business departments.

- *Analyze application-domain implications. Phase: Requirements Analysis. Mentioned: 2 times.* Architects encountered the importance of analyzing application-domain implications on satisfying requirements at the early project phases. Often the critical issues, known only to application-domain experts, are hidden in the requirements.

- *Analyze and negotiate cost-vs-importance tradeoffs. Phase: Requirements Analysis. Mentioned: 2 times.* In some cases, realization of certain requirements lead to substantial increase of system complexity and development cost. An architect should be able to identify such requirements, evaluate their importance and report them to stakeholders.

- *Analyze technology implications. Phase: Architecture Design. Mentioned: 2 times.* For time-critical systems, the implications of deploying any kind of technology on the system performance are often hidden in the beginning of a project and sensitive at the end of a project. Therefore, an architect should analyze possible implications already at the design phase. One of the architects gave an example of such implication: a Java image processing library, deployed for a medicare scanner, showed insufficient performance at the latest project phases, which led to partial software re-design.

- *Avoid preemptive scheduling. Phase: Architecture Design. Mentioned: 2 times.* Preemptive scheduling features more efficient processor usage, but reduces task-latency predictability and increases number of the context switches.

- *Add margins for performance. Phase: Architecture Design. Mentioned: 1 time.* Architecture over-dimensioning helps to reduce risks related to performance failures, though this may increase system complexity and cost.

- *Assign budgets and avoid parallel processing. Phase: Architecture Design. Mentioned: 2 times.* Assigning processor or bus budgets for real-time tasks removes the task interleaving and blocking problems.

- *Put critical tasks at lower layers. Phase: Architecture Design. Mentioned: 1 time.* For layered architectures, a task execution flow propagates through many layers, which causes significant performance overhead. Putting time-critical tasks only at lower layers (hardware drivers, operating system and middleware) shortens the task execution path and removes unpredictable fluctuations in performance.
- *Operational modes. Phase: Architecture Evaluation. Mentioned: 2 times.* An operational mode describes a stable execution configurations of a system. The architect also stresses the importance of considering the transitions between the operational modes, because they often pose performance problems, like high processor load and task-latency delays.
- *Analytical and simulation-based evaluation. Phase: Architecture Evaluation. Mentioned: 3 times.* Among various evaluation techniques proposed in literature, the architects favor analytical and simulation techniques due to their simplicity and short learning curve.
- *Apply worst- and average-case loads. Phase: Architecture Evaluation. Mentioned: 2 times.* This approach considers variations in environmental inputs to a system. Considering average loads from the user, environment or other systems helps to understand if the system is over- or under-dimensioned. Worst-case loading enables the detection of possible performance bottlenecks.

Summarizing this subsection, architects face vague/hidden requirements, unpredictable hardware and operating system behaviour. Besides this, they have to consider new technologies of which the the influence on the requirements is unclear. The most challenging properties to address are task latencies and throughput. The architects encounter various approaches and techniques to deal with the above problems. The scenario-related approaches have been mentioned by all six architects, even prior to the discussions on scenarios in the interview flow.

8.3.3 Scenario Identification and Analysis

As we have mentioned in the previous subsection, all of the architects deploy scenarios in their architectural approaches. This subsection explains in detail how the architects identify and analyze scenarios.

Table. 8.6 shows how frequently the architects deploy scenario-based analysis in practice.

Four out of six interviewees reported direct usage of scenarios in every project. All the interviewees noted that they have never explored the complete state space of a system (all execution configurations) in their experience.

Table 8.6: *Frequency of using scenarios for architecture analysis.*

In your architectural practice, how often do you use scenarios for realization and evaluation of performance properties?						
Interviewee	**1**	**2**	**3**	**4**	**5**	**6**
From Never to Every time	Every time	Every time	Often	Often	Every time	Every time

A. Scenario identification

There are no commonly used semi-automated methods and tools for scenario identification reported in the literature. Hence, architects are still using their own approaches, experience and intuition to find the set of right scenarios for system analysis. The interviewees encountered a number of approaches deployed in their practice. Fig. 8.6 summarizes the approaches and groups them into the following fours clusters: software issues, hardware issues, domain knowledge and past experience.

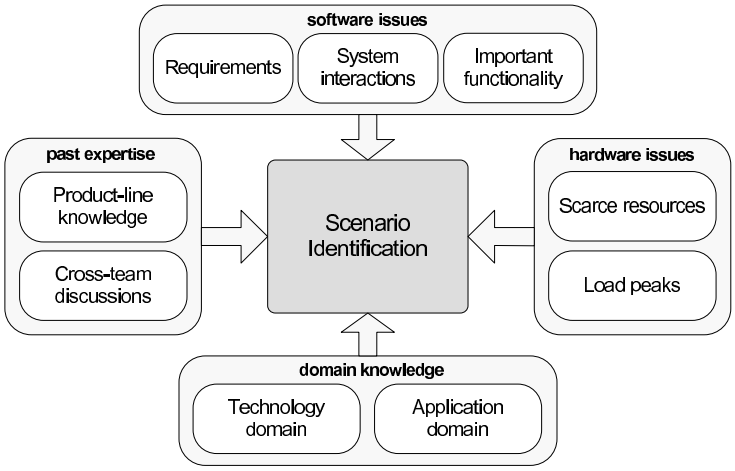

Figure 8.6: *Approaches for identification of scenarios for further analysis.*

Below, we present explanations given by architects for each of the scenario identification approach.

- *Product-line knowledge.* Product-line architectures are characterized by similarities in requirements and design. If a currently designed system is a next entity in a product-line family, an architect can reuse the scenarios from the previous projects.

- *Cross-team discussions.* Project teams have expertise in different problem and technology domains. Therefore, open talks with customers, analysts, test-team and even with end-users help to identify scenarios initially hidden from an architect's attention. Such discussions help to make the scenario set more representative for further analysis and reduce the risk of unexpected problems.

- *Requirements.* One of the most straightforward and reliable ways to identify scenarios is to look at performance requirements. Normally, a performance requirement defines a stimulus to a system (e.g. a user action) and a latency interval, in which the system should process the request and respond to it. Such a requirement already represents a simple scenario. Combining the performance requirement helps to generate more complex but still relevant scenarios.

- *System interactions.* Scenarios can be obtained by analyzing the system's interaction with the outside world (users, environment and other systems). Any stimulus from the outside world causes at least one execution trace in a system. One stimulus or a number of such stimuli (if they happen in parallel) can define a scenario relevant for consideration.

- *Important functionality.* An architect selects important or frequently used functions from the functional requirements and analyzes possible execution scenarios that represent these functions. The scenarios can be combined, if the functions they represent may happen in parallel.

- *Critical hardware resources.* Scarce hardware resources always capture attention of system architects, since they often cause performance deterioration. Analysis of occupation of such critical resources enables identification of software entities (objects, messages and tasks) that use the resources. The scarce hardware resources often block the software entities execution. Therefore, the execution scenarios, in which these software entities take part in, are always relevant scenarios for performance analysis.

- *Load peaks.* Hardware resource overload situations (e.g. a PDA processor is occupied by 100%, while decoding an MPEG video I-frame) may significantly reduce performance of other software entities that use this hardware resource at the same time. Therefore, an architect should consider scenarios that model these overload situations.

- *Technology domain.* In some cases, an adopted technology causes hidden problems for the system performance. A common example is the Java garbage collector. In order to reduce the risk of such problems at the early development phases, an architect may try to encounter and analyze

those scenarios that extensively exploit that adopted technology. For instance, for a client-server architecture deploying CORBA technology, it is worthwhile to consider scenarios with a heavy load from client messages to a server.

- *Application domain.* The interviewees pointed out that application-domain analysis is an important factor for successful design of any kind of system. With respect to performance-critical systems, the application-domain analysis helps answering the following questions: (a) what can go wrong in a system environment that may lead to system failures or deterioration in the system service, (b) how does the environment trigger activities in a system, and (c), what kind of response should the system give back to the environment?

The architects described the above-mentioned approaches for identification of general or common scenarios. In this chapter, we also focus on methods for identification of *critical* scenarios.

B. Usage of critical scenarios

This subsection is about the architect's use of critical scenarios in the performance analysis. First, we discussed with the interviewees the following two issues: (a) the average number of scenarios used for analysis in one project, and (b) the proportion of critical scenarios in the total amount of considered scenarios (see Table 8.7).

Table 8.7: *Empirical data on frequency usage of critical scenarios.*

ID	**1**	**2**	**3**	**4**	**5**	**6**
How many scenarios do you normally consider for analysis?						
	>10	2-5	2-5	5-10	1-10	>10
Do you consider all scenarios or do you select only critical scenarios that may jeopardize performance requirements?						
	Only critical	Only critical	Only critical	Critical and non-critical	Critical and common	Only critical

The answers show that four out of six architects consider only a few scenarios (from 1 to 10) for the analysis. The other two architects reported that they consider as many scenarios as needed to cover the requirements. However, all architects mentioned that they consider mostly critical scenarios for analysis. During the interview they acknowledged the awareness that, analysis of a limited subset of scenarios does not guarantee completeness of the performance

assessment results. The architects pointed out that due to the time pressure they always focus only on critical scenarios. The architects reported that the best way to handle the project risks imposed by limiting to critical scenarios only, is to document and monitor these risks during the following phases of the project.

C. Approaches for selection of critical scenarios

The correct selection of critical scenarios from the set of all scenarios is vital for obtaining accurate performance predictions. It also helps to reduce the risks of unexpected failures to meet performance requirements at the later project development phases. During the survey, we have collected the approaches for critical scenario selection employed by the architects in their industrial practice. Fig. 8.7 outlines these approaches.

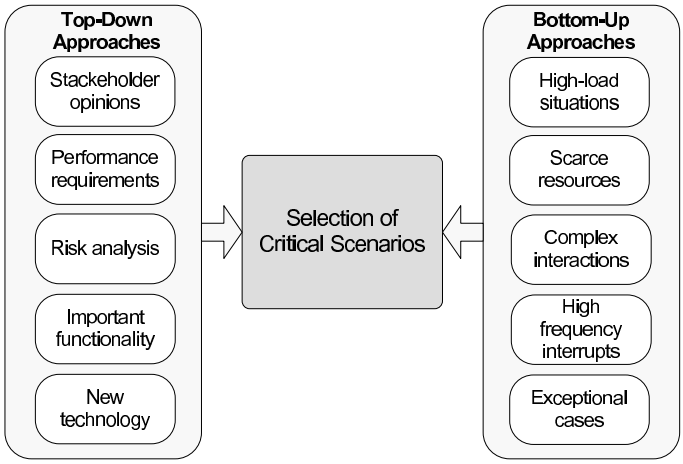

Figure 8.7: *Approaches for selection of critical scenarios.*

The survey has revealed two complementary types of selection approaches: *top-down* and *bottom-up*. A workflow of the top-down approaches starts from the analysis of high-level issues (risk analysis, requirements, etc.) and proceeds to lower levels of considerations (static architecture, system behaviour and interactions). The bottom-up approaches begin with investigation of fine-grained architectural issues (hardware interrupts) that might cause performance problems and proceed with tracing those issues to general system behaviour. The following paragraph describes both types of approaches in more detail.

- *Stakeholder opinions.* Stakeholders are normally experts in their application domain. They can identify risks and potential performance problems based on previous experience in the development of the same type

of systems. The task of an architect is to collect these problems from the stakeholders, analyze and convert them into architectural terms, i.e. execution scenarios. For example, a stakeholder of a DTV system may share his concern on channel zapping functionality, which was performing too slow in his previous product. In this case, an architect identifies the "zap a channel" scenario as a critical one for later analysis.

- *Important functionality.* Important or critical functionality can be analyzed based on prioritized functional requirements. High-priority requirements normally show which functions are most important for successful system operation. An architect may describe these functions as execution scenarios for further task latency analysis.

- *Performance requirements.* Most of the performance requirements are structured in the following way: "upon a stimulus X, a system should complete an action A in N time units". This type of requirements directly represent an execution scenario specifying the input and system activity to be performed. Moreover, this scenario is critical because it should be executed in a timely manner. Combination of a set of performance requirements can also represent a critical scenario, if a system obtains a set of stimuli and executes actions in parallel.

- *Risk analysis.* Another source for critical scenarios selection is a risk analysis specification. Risks related to technical aspects can be modeled as a scenario, in which a stimulus is extracted from the list of undesired inputs to the systems.

- *New technology.* Performance aspects of a new technology should be checked prior to deployment. All possible system configurations and behaviour related to or involved in the technology, are good candidates for thorough analysis against performance requirements.

- *High-load situations.* Scenarios, in which a system receives a worst-case input from the environment to be processed, are relevant for performance analysis due to the possible artifacts like cache misses, bus/processor blocking, memory overload, etc. For the specification of such high-load scenarios, it is important to accurately identify worst-case boundary values for input stream or stimulus parameters.

- *Scarce resources.* Scarce hardware and software resources may be blocked or overloaded during a system execution. Blocked or overloaded resources may delay a time-critical task execution in an unpredictable way. Therefore, the architect needs to: (a) identify a subset of scarce resources, and (b) determine the scenarios that use these resources for execution.

- *Complex interactions of tasks.* Parallel execution of tasks or task dependencies impose high variations in the task completion times. Therefore, our architects proposed selecting scenarios in which a lot of task communication and interleaving occur. These scenarios help to discover possible performance bottlenecks in a system.

- *High-frequency interrupts.* This approach suggests the following actions: (1) identify hardware drivers with high-frequency interrupts, (2) trace tasks that are triggered by these interrupts and, (3) specify execution scenarios in which these tasks are involved. These actions help to find and analyse resource overload situations, because frequently interrupted drivers may impose a high load on processing and communication resources.

- *Exceptional cases.* Architects pointed out the importance of searching for unspecified environmental inputs and internal configurations. An architect should ask himself "What can go wrong in the system or in its environment?". Scenarios representing these exceptional situations, are relevant for throughput, task latency and resource load analysis.

As the reader may have noticed, the set of approaches is quite diverse and requires not only knowledge and experience from an architect, but also communication and business analysis skills. Besides this, most approaches assume availability of well-written documents on requirements, risk analysis, deployed technology and software/hardware architectural blocks. In practice, these documents are only partly available in the industry.

The next two parts in the survey, described in the following subsections, revealed the challenges that the architects experienced during selection critical scenarios.

D. Challenges in selection of critical scenarios

Fig. 8.8 depicts an overview of the challenges in selecting critical scenarios.

The interviewees admitted that the most challenging issue for scenario selection is a *combinatorial explosion of scenarios* in complex systems. For instance, the number of possible scenarios for a set-top-box system is in the order of magnitude of 100,000. Extraction of scenarios relevant for performance analysis is a non-trivial task.

Requirements which are changing over development phases and even after the product release, are another obstacle for correct scenario selection. A requirement can evolve over time and convert a scenario to a critical one. An architect should be able to forecast possible changes in the requirements, based on current trends, in order to select scenarios that would reflect also the requirements evolution.

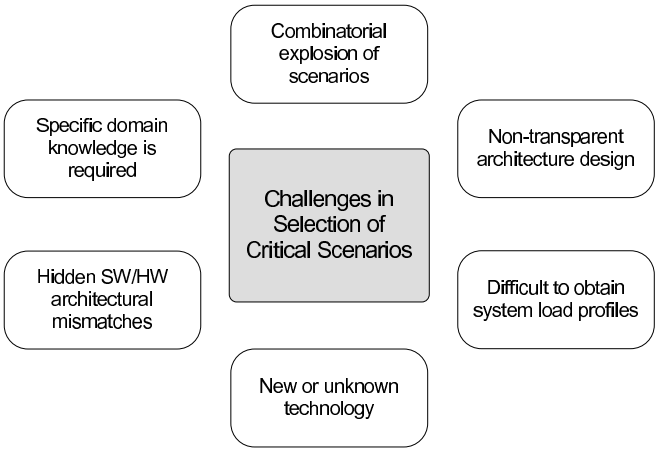

Figure 8.8: *Challenges in identification and selection of critical scenarios.*

As we mentioned above, application-domain knowledge is one of the sources for scenario identification and selection. However, an architect is not always an expert in a certain application domain. He can only collect information on the relevant scenarios for a particular domain from the stakeholders or business people. However, that imposes the following assumptions: (a) the people providing this information are available, (b) the architect interpretation of the information is correct.

Another challenge appears once a *new or unknown technology* is deployed for a product. In this case, there is no human-based source of information available for the architect. Relevant data can be obtained either remotely from Internet, or from technology documentation. Unfortunately, these two sources do not guarantee a complete overview on the implications of the technology deployment onto a certain system.

Non-transparent architecture or design of individual modules has been mentioned also as an obstacle for correct selection of critical scenarios. Incomplete documentation and non-structured static and dynamic architectures complicate the following aspects: (a) understanding of scenario behaviour resulting from a certain input and (b) tracing the behaviour through system modules. In other words, it precludes specification of scenario structure and behaviour and limits the comparison/selection possibilities.

Last but not least, an important challenge mentioned by architects is the *analysis of system-load profiles*. These load profiles specify various inputs to the system from its environment and hardware interrupts. These inputs are normally characterized by input frequency distribution, size of transferred data and the system entity receiving that input. However, for some systems that

have a heterogeneous or context-dependent environment, these load profiles cannot be obtained in a straightforward manner. This restrains the correct description of stimuli/triggers for a scenario.

E. Critical scenario types most difficult to identify

The architects have been asked for their opinion on the scenarios that are most difficult to identify. The scenarios mentioned in the answers have been categorized into six types. Table 8.8 summarizes the reported types of critical scenarios that are difficult to identify and ranks them by the popularity in the interviewee answers.

Table 8.8: *Most problematic types of critical scenarios to identify.*

What types of critical scenarios are most difficult to identify?	
Number of References	**Scenario Type**
5	**Related to hidden resource utilization**
4	**Caused by environment and end-user**
4	**Caused by platform events**
2	**Related to future changes in a system**
1	**"What can go wrong" scenarios**
1	**Context-dependent scenarios**

Five out of six architects consider that the scenarios related to a *hidden resource utilization* are most challenging for identification. A hidden usage of resources is not specified in design documentation and non-transparent to an architect. While the hidden utilization may substantially reduce the expected performance, it is often discovered only at the later testing phases of a project, when the real execution profiles can be obtained.

Scenarios caused by the environment and end-user were marked as hard to identify due to lack of user studies, prototype tests in real environment. Scenarios triggered by hardware platform events can be identified from hardware/driver specification. However, a comprehensive specification is rarely available.

Two out of four architects mentioned that scenarios related to *future changes* are hard to determine. The main reason is a system evolution path which is difficult to predict. The same prediction problem holds for scenarios of the type “What can go wrong”.

Finally, one architect mentioned *context-dependent scenarios.* Current sys-

tems tend to become dependent on the context, in which they operate. Dynamic systems may operate in hundreds of different contexts. Besides this, these contexts can be combined, so a system operates in a number of contexts simultaneously. In this case, identification of context combination, which would impose a threat to system performance, is non-trivial task for an architect.

F. Missing critical scenarios and consequences

In order to analyze the consequences of the scenario identification challenges discussed above, we have asked the architects to reflect on how frequently they failed to identify the complete set of critical scenarios in practice. Table 8.9 shows the interviewee answers in the range from *never* to *always*.

Table 8.9: *Frequency of critical scenario identification failure.*

In your architectural practice, how often did you fail to identify complete set of critical scenarios?						
Interviewee ID	**1**	**2**	**3**	**4**	**5**	**6**
From Never to Always	Often	Often	Rarely	Rarely	Rarely	Rarely

The interviews showed that an accurate identification of critical scenarios is a problem. One third of the interviewees admitted that they regularly fail to identify the complete set of critical scenarios, while the rest reported that this happens rarely. Table 8.10 describes the impact of missing critical scenarios onto product development time and market success.

The consequences resulting from missing the critical scenarios have a broad character, starting from several hours of system operation-stop, proceeding to missing of market shares, and ending with a project cancelation. The following project cancelation example has been given by one of the architects. A performance requirement for a database conversion in an expert system specified a maximum duration of 8 hours. First tests showed that the designed software and hardware architecture completes the conversion in 300 days. This scenario was not selected as a critical one and no further analysis was performed. Later analysis encountered that the problem was in the disk access time, so that the project was finally canceled.

Table 8.10: *Consequences of failing to identify a critical scenario.*

What was the impact of omission of critical scenarios onto product development time and market success?	
Interviewee ID	**Impact Example**
1	**Project cancelation**
2	**Unsatisfied customers**
3	**Missing market share**
4	**NA**
5	**Project cancelation**
6	**Several hours of system shut down**

8.4 Justification of Scenario-Based Approaches

This subsection provides a comparison between the following two orthogonal modeling and analysis approaches: scenario-based and full search. The *full-search approaches* require modeling and analysis of all possible system configurations and behaviour traces, while the *scenario-based approaches* suggest specification and assessment of only relevant (important) execution and configuration scenarios. We have asked the architects to evaluate both approaches against various characteristics, such as modeling and analysis efforts, analysis completeness and complexity, scalability and risk level. Table 8.11 presents the comparison results. A sensitivity graph of the major three characteristics (efforts, completeness and risk level) is also depicted in Fig. 8.9. Based on the average of the curves, provided by the architects, each of the curves was normalized to a unity interval, so that they can be jointly visualized.

The interviewees favored scenario-based approaches in terms of relatively low *modeling* and *analysis efforts*. They save time and resources, which in an industrial environment is a substantial advantage. Some architects pointed out that deployment of full-search methods is not feasible for complex systems, due to time pressure and unavailability of human resources. Moreover, complete specification and analysis requires addressing many details that are irrelevant for architectural issues. One of the architects even expressed that *it would be a complete waste of time and resources to analyze all possible behaviours and configurations of a system.*

However, full-search approaches provide *analysis completeness* and higher accuracy of the predicted performance values. This results from the involved exhaustive modeling. Moreover, the exhaustive search reduces *risks* of unex-

Table 8.11: *Comparison of the scenario-based and full-search approaches.*

Property	**Scenario-based approach**	**Full-search approach**
Modeling efforts	Medium	High
Analysis efforts	Low	High
Analysis completeness and prediction accuracy	Medium	High
Risk level	High	Low
Usage complexity	Low	High
Value for hard real-time systems	Medium	High
Value for soft real-time systems	High	Low
Value for large systems	High	Low
Scalability for evolving architectures	High	Medium

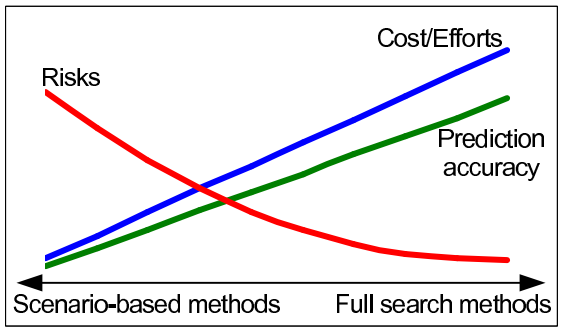

Figure 8.9: *Characteristics of the scenario-based and full search approaches.*

pected performance problems at the later development phases or even during the product deployment. Scenario-based analysis does not provide completeness, and therefore, requires accurate risk management policies.

The next reviewed property was *usage complexity* of the two method types. The survey showed that full-search approaches (especially, those involving formal methods) have a high deployment complexity and a flat learning curve. In order to achieve (or even prove) completeness of obtained results, a full-search analyst should be able to accurately operate with corresponding mathematical instrumentation (lemmas, theorems, proofs, etc.).

Another interesting comparison viewpoint of the methods is the efforts vs. accuracy trade-off imposed by both method types. Fig. 8.10 depicts the comparison analysis of the scenario-based and full-search approaches in terms of their efforts vs. accuracy trade-off. Based on the average of the curves, provided by the architects, each of the curves was normalized to a unity interval (this also holds for the subsequent figures). The interpretation of the obtained curves is as follows. Even when aiming at a low accuracy of performance prediction (from 0 to 40%), the full-search methods require a substantial amount of modeling and analysis effort. When striving for higher accuracy (up to the level of 100%), the amount of required effort grows relatively slowly. In contrast, the scenario-based methods require low effort for obtaining performance predictions of low and medium accuracy. However, once the required prediction accuracy reaches the level of 90-100%, the number of scenarios to analyze substantially increases, and therefore, the effort grows exponentially. As a result, for analysis of safety-critical systems requiring very high prediction accuracy, it is more valuable to deploy one of the full-search approaches.

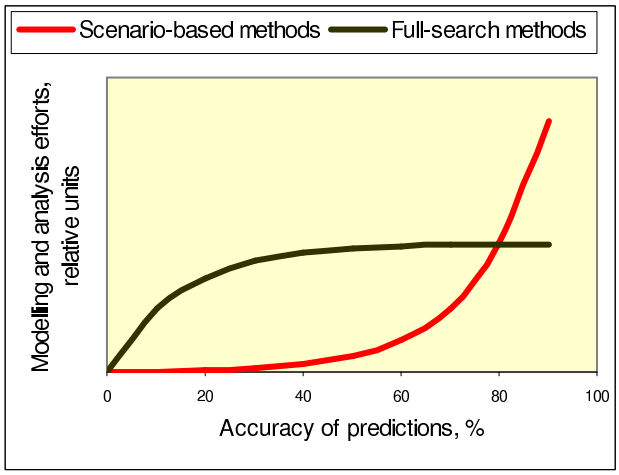

Figure 8.10: *Effort vs. accuracy trade-off for full-search and scenario-based methods.*

An interpretation of this efforts vs. accuracy trade-off is given in Fig. 8.11, which represents the value of scenario-based approaches applied for analysis of soft and hard real-time systems. The horizontal axis of the chart depicts the percentage of considered scenarios from all possible scenarios. The vertical axis shows the value of applying a scenario-based approach. The value is measured in abstract units starting from a *null* value. The chart shows that the value of applying scenario-based approaches for soft real-time systems grows fastly already from a low percentage of scenarios analyzed. The value curve saturates

at the level of 30% from all possible scenarios. Further increase in the amount of the considered scenarios adds little value for assessment, because soft real-time systems do not require completeness and high accuracy in predictions.

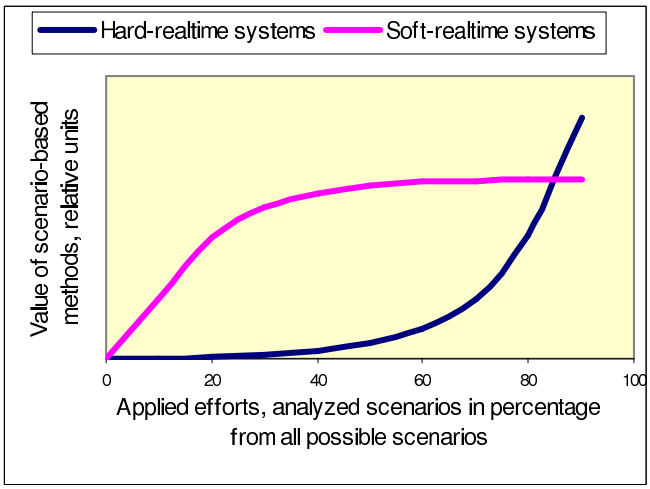

Figure 8.11: *Applicability of scenario-based approaches for soft and hard real-time systems.*

For hard-real time systems, the trend is opposite. Due to the fact that such systems require accurate and complete predictions, the low amount of analyzed scenarios brings no value for performance analysis. In order to reach analysis completeness, the amount of considered scenarios should be close to the 100% bound. The Fig. 8.11 implies a conclusion that scenario-based approaches are valuable for soft real-time systems and hardly applicable to hard real-time and safety-critical systems.

The next comparison criterion for the two types of methods is the involved modeling and analysis effort related to the scale and dimensionality of the problem. The problem scale relates to the complexity of a system to be designed (number of extra-functional requirements, software and hardware building blocks, computational intensity). The data obtained from the architects is visualized in Fig. 8.12. The chart shows dependencies between the problem size and the amount of effort needed for problem modeling and analysis in full-search and scenario-based approaches. For small-sized problems, both types of approaches require low efforts. However, once the problem size increases, scenario-based approaches feature proportional effort increase, while full-search approaches feature exponential increase in modeling and analysis efforts. The chart shows that the full- search paradigm is hardly applicable to the analysis of very large and complex industrial systems.

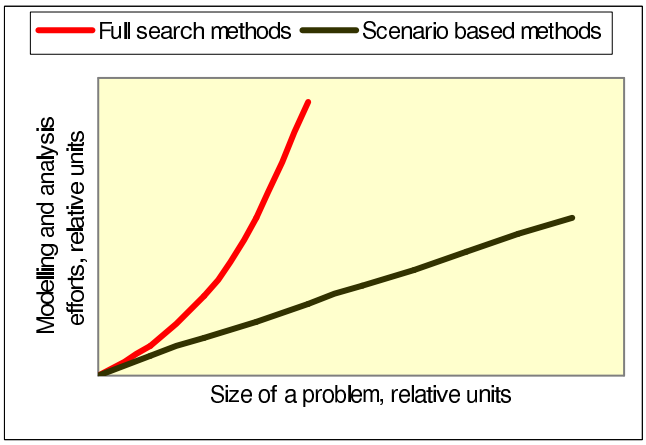

Figure 8.12: *Dependency between amount of efforts and problem size for scenario-based and full-search approaches.*

The interviewees acknowledged that in practice, for complex systems, the scenario-based approaches are used for global analysis, while the full-search approaches are applied to safety-critical submodules.

Let us now discuss the *scalability for evolving architectures* aspect from Table 8.11. Scenario-based approaches have higher scalability, due to their modeling compactness and focus on specific operational modes. For instance, a change in a system configuration would require adjustment and re-assessment of related scenarios only. For full-search techniques, it is necessary to change and solve the complete formal system model.

The architects also mentioned the side benefits of using scenarios for other phases of a project. Scenario-based methods help with: (a) stakeholders to express their needs, (b) requirements experts to elicitate and specify requirements, (c) architects to evaluate architectures and, (d) testers to identify proper test cases. It is important to mention that for these project phases the same set of scenarios can be used.

Summarizing this subsection, we state that, although scenario-based approaches are widely used in industrial practice, they also impose a number of trade-offs for an architect. Usage of scenarios does not guarantee completeness of analysis and, therefore, introduces technical risks of failing performance requirements. For this reason, scenarios are hardly applicable to hard real-time and safety-critical systems. Another drawback resulting from using scenarios is the challenge in identification and proper selection of critical scenarios for analysis. Despite of the above-mentioned disadvantages, scenario-based methods feature the following important benefits: (a) low effort for modeling and

analysis, (b) relatively low usage complexity, (c) high applicability for large and complex systems, and (d) high scalability for evolving architectures.

8.5 Conclusions on the Survey

This chapter has presented a survey that we have performed among a group of experienced architects from different application domains in the industry. The objective of the survey was to obtain empirical data and practical knowledge on performance assessment and architecture evaluation using scenarios. The survey consisted of two parts: written questionaire and face-to-face interviews.

The survey has revealed that scenarios are widely used in industry for assessment of performance-critical systems. All six interviewed architects mentioned that they use scenarios in practice. Four out of six architects reported that they deploy scenario-based approaches only for architecture performance assessment. However, the interviews also showed that scenarios are not a "silver bullet" solution in architecture assessment, because they have clear advantages and drawbacks. The opposite paradigm to the scenario-based analysis is a full-search analysis of system states and behaviour, which is often implemented by formal methods. A comparison between these two orthogonal paradigms enables drawing of a number of conclusions.

The major benefits of the scenario-based approaches are the following:

- reduction of time, cost and effort needed for performance assessment,
- focus on relevant performance issues only,
- human-readable specification of system interactions (inputs and outputs) with the environment, as well as its internal behaviour.

The main drawbacks of using scenarios include the following:

- missing a scenario representing performance-critical interaction or behaviour trace of a system may lead to performance failures at the product deployment phase,
- identification of scenarios requires substantial experience from an architect in the application and technology domains.

The comparison of the scenario-based and full-search analysis paradigms introduces clear cost-vs-completeness trade-off. All the interviewed architects reported that, in the industrial environment, due to the time-to-market pressure, the trade-off is solved in favor of saving costs and efforts, i.e. for scenario-based analysis. However, for safety-critical system domains, such as defense, space, automotive and medical systems, this trade-off should be solved in favor of completeness, i.e full-search analysis.

The interviews and literature search also uncovered that there are no well-defined scenario-based methods deployed in the industry. The architects apply

scenarios using their own knowledge and past experience. The chapter describes a number of hands-on approaches for identification of scenarios and later selection of critical scenarios for further analysis. Moreover, the chapter presents common challenges that the architects experienced in using scenarios, and the consequences when these challenges are not addressed in a correct and accurate way.

The last section in this chapter is dedicated to justification of scenario-based analysis in an industrial environment and reports on various trade-offs between the two orthogonal paradigms: scenario-based and full-search approaches. The survey shows that scenario-based approaches do not guarantee completeness of analysis and may impose technical risks of failing performance requirements. Moreover, an architect needs substantial knowledge and experience in order to identify and select the correct scenarios for analysis. Nevertheless, scenario-based methods are a proper (and, sometimes, the only single) choice for analysis of complex systems, due to their low modeling and analysis complexity, relatively low usage complexity, and high scalability for evolving architectures.

Chapter 9

Case Studies

9.1 Introduction

The first objective of this chapter is to describe the validation of the DeepCompass framework. The validation involves the design and development of the three real-time applications: MPEG-4 Decoder, Car Radio Navigation (CRN) System and JPEG Application. Apart from the validation of the performance prediction techniques, we have also applied the architecture optimization to the CRN and JPEG systems. The goal of the optimization of Chapter 6 was to identify optimal solutions from a set of generated architectural alternatives and to find out the guidelines for further improvements of these alternatives.

The second objective of this chapter is to help the reader in better understanding of the modeling details and working principles of the scenario-based performance prediction method. Besides this, the reader can trace our way of reasoning applied in practice through all phases of the DeepCompass framework.

It is important to mention that these case studies have been carried out during the development of the framework. The hands-on experience, obtained from these case studies, has helped the author to correct and improve the framework both from the process view and from the technical view.

The chapter is structured into the following sections. In Section 9.2, we describe the MPEG-4 Decoder case study, which aims at a portable PDA with video object-oriented processing. Section 9.3 discusses the case study on the CRN system, which serves as a benchmark for performance analysis methods.

Section 9.4 presents the results of our case study on the JPEG application, employing active components. Section 9.5 concludes the chapter and indicates results on the effort spent for each of the case studies.

9.2 MPEG-4 Decoder Application

After the first iteration of the development of the DeepCompass framework, we have validated it with first case study on a state-of-the-art MPEG-4 coding application. We have used the full specification of the standard, featuring arbitrary-shaped video objects. Applying the scenario-based approach which is a cornerstone of the DeepCompass framework, we have predicted the performance and real-time properties of the designed decoder. After the integration phase, we have deployed the decoder on the Linux/Intel platform and have measured actual performance data. The accuracy of predictions is obtained by comparison of the predicted results with the real execution data.

9.2.1 MPEG-4 Decoder Functionality

Let us now provide some details of advanced video object-oriented processing. Fig. 9.1 depicts the computation graph for arbitrary-shaped video object decoding as used in MPEG-4 [83]. Similar to MPEG-2, objects are divided into macroblocks (MB). The diagram shows special processing stages for decoding the shape and motion of the video objects, in addition to the usual texture decoding. Each decoding job iteration starts with macroblock type decoding (MBtype Dec). The ShapeMC stage computes the motion compensation for the Shape part and provides referenced MBs for the Context Arithmetic Decoding (CAD) stage. The CAD provides an MPEG-compliant shape representation of the macroblock. The shape for the macroblock is represented by a 16×16 binary sub-image, which is sent to the outputs of the Shape job. The Coded Block Pattern (CBP) extracts information about parts of the texture that need to be updated.

Texture decoding involves five steps: Motion vector Decoding (MvD), IDCT Coefficient Decoding (Coeff Dec), Texture Motion Compensation (TextureMC), Inverse Quantization (IQ) and Inverse DCT (IDCT). The MBtype Dec, CAD, CBP, MvD, and Coeff Dec stages are executed sequentially, because each stage depends on another stage, to specify the next position in the input bit stream. Therefore, we introduce a loop surrounding these stages, indicating the order between them. In [83] we discuss the decoder structure in detail.

Having defined the MPEG-4 logical processing blocks and their communication, we follow the proposed DeepCompass framework to build the coding application with predictable performance.

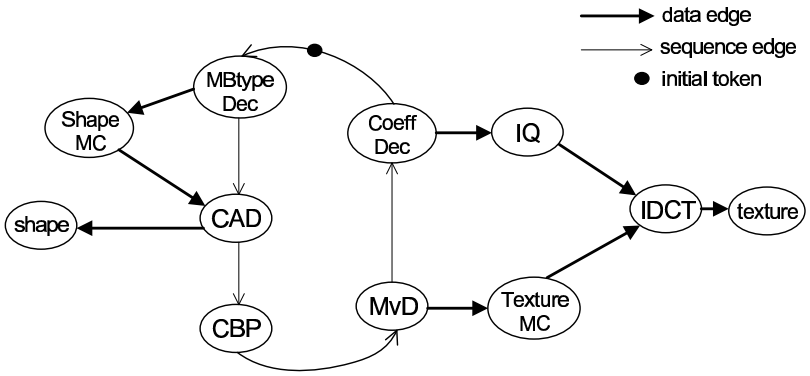

Figure 9.1: *Computation graph of arbitrary-shaped MPEG-4 video object decoder. The nature of the arrows indicates their purpose.*

9.2.2 Component specification

In this section, we distinguish the individual components and specify them independently, including their interfaces. To satisfy the functional requirements of the application, we have developed four ROBOCOP CBA components: Reader, Buffer, Decoder and Renderer (see Fig. 9.2). For simplicity of explanation, each component contains one service with the same name. The Reader service has an `IRead` provided interface (implementing `readFrame()` and `startReadingThread()` operations) and an `IBufferAccess` required interface. The Buffer service has only an `IBufferAccess` provided interface, implementing buffering operations `popElement()` and `pushElement()`. The Decoder service provides an `IDecode` interface with operations `decodeFrame()` and `startDecodingThread()`. In addition, the Decoder service requires two buffers for operation via its `IBufferAccess` interface. Finally, the Renderer service provides an `IWrite` interface with `writeFrame()`, `startWritingThread()` operations and requires `IBufferAccess` interface.

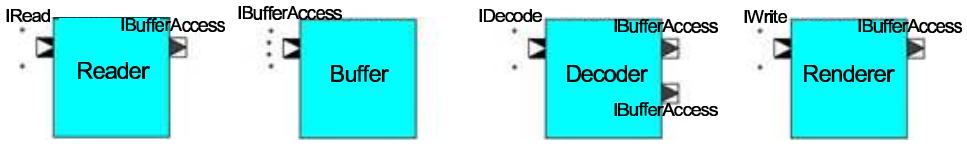

Figure 9.2: *Components developed for the MPEG-4 decoder application.*

Each component is accompanied with its corresponding *resource* and *behaviour* models (see a simplified version in Fig. 9.3). The resource model specifies resource requirements per individual service operation, while the be-

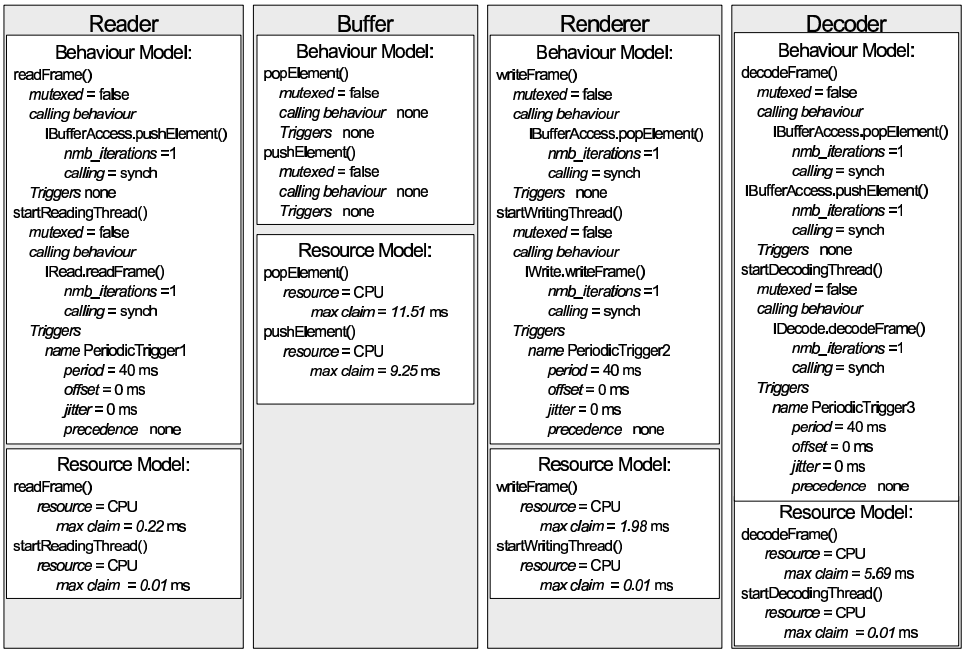

Figure 9.3: *Behaviour and resource models of the developed components.*

haviour model also describes the underlying calls to other operations per service operation as well as thread triggers (if existing) implemented by this operation. The resource-usage data per operation has been extracted by testing and profiling of each individual service. The operation behaviour data has been generated from the service source code. Reading the Decoder model as an example, we can see that the operation `IDecode.decodeFrame()` first calls `IBufferAccess.popElement()` operation (takes encoded frame from the buffer), then decodes the frame in the core of the `decodeFrame()` and finally calls `IBufferAccess.pushElement()` to store the decoded frame to a buffer. All calls are synchronous. The maximum CPU claim of the operation `decodeFrame()` itself equals 5.69 ms. Note that the CPU times (indicated as 'claims') of called `pushElement()` and `popElement()` operations are specified in the Buffer model, being 9.25 ms and 11.51 ms, respectively[1]. Another example is the `startDecodingFrame()` operation. It implements a timer which periodically (40 ms) triggers the operation core execution. This means that once the operation is called, the firing timer is created and the operation core will be executed periodically.

[1]The relatively high processing claim of the buffering operations is explained by the prototyping implementation, where the storage and retrieval functions are pixel-based.

9.2.3 Component Assembly and Scenario Identification

For the scenario identification, the services are graphically composed by means of the CARAT toolkit into a component-based MPEG-4 application. The composition process consists of two activities: (a) instantiation of services and (b) binding the instances via their interfaces (see Fig. 9.4). As indicated by the figure, the Reader instance is bound to Buffer1 to store the encoded video frames. The Decoder instance is bound to Buffer1 to read these frames, and to Buffer2 to store the decoded pixels. The Renderer instance is bound to Buffer2 to read the decoded pixels and render them on the display.

After composition, a *critical scenario* is selected. We predefine a normal execution mode with a resolution of 340×280 pixels and a frame rate of 10 frames/s as a critical scenario. Furthermore, a *scenario model* is specified for the chosen execution configuration (see Fig. 9.5). It consists of an *assembly structure* and a number of control inputs (*stimuli*). Those inputs (event, periodic timer, interrupt, etc.) lead to the frame-periodic execution of one of the component operations. In our case, we have designed three application-level periodic timers that call `Reader.readFrame()`, `Decoder.decodeFrame()` and `Renderer.writeFrame()` operations with a periodicity of 100 ms, thereby establishing the pipes-and-filters execution architecture. Note that we specify a *deadline* for each task instance triggered by a stimulus (also 100 ms). These are the *real-time requirements* for the system that we are going to validate in the later simulation and analysis phases.

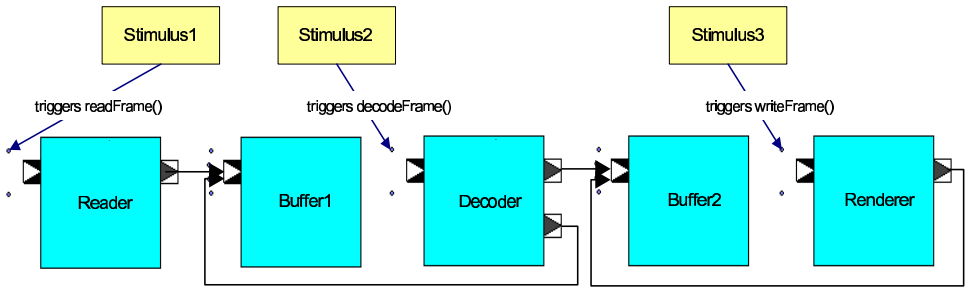

Figure 9.4: *Component assembly of the MPEG-4 decoder.*

9.2.4 Model Synthesis and Task Generation

The CARAT preprocessor synthesizes all corresponding component resource and behaviour models together with the scenario model in the executable system model. The latter model details the set of tasks running in the scenario. A *task* is defined by the following parameters: time period (or minimal inter-arrival time, if aperiodic), deadline, offset, precedence constraints with other

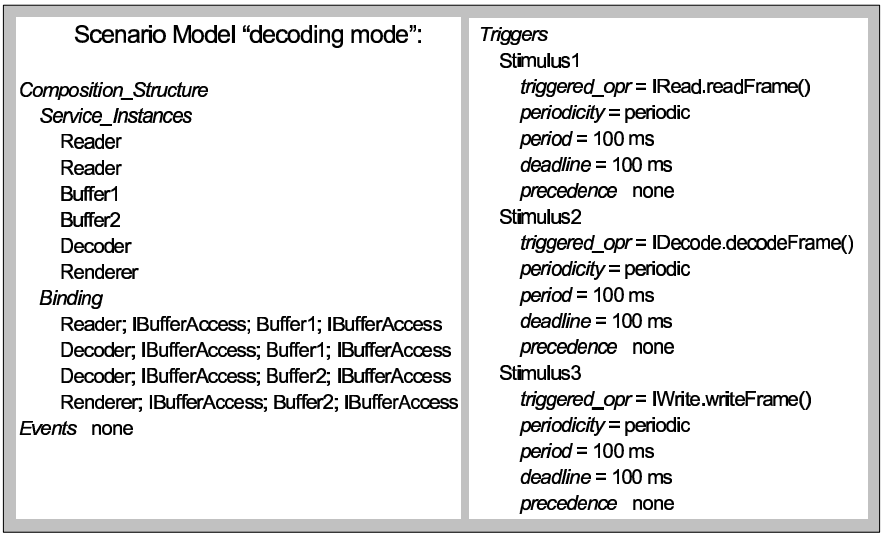

Figure 9.5: *Specification of the scenario model. The scenario plays MPEG-4 video with a rate of 10 frames/s.*

tasks, and its sequence of operation calls through components made by each task instance. Except for the last parameter, all parameters are inherited from the stimulus parameters in the scenario model. The task call sequences are reconstructed from the individual call sequences of constituent operations specified in the behaviour models. The processor load of a constituent operation is known from the corresponding resource model, so that the CARAT tool can calculate the total execution time of a task instance. The CARAT visualizer draws the generated tasks (see Fig. 9.6) to help in understanding and analysis of the task behaviour in the scenario.

For instance, the decoding task (see Fig. 9.6(c)), triggered by Stimulus2, contains three operation calls of the Decoder, Buffer1 and Buffer2 service instances. The CPU utilization times of each called operation are known from the corresponding resource model (see Fig. 9.3). The accumulated total execution time of this task involving the three called operations is 26.25 ms. The period and deadline of the task (100 ms) are derived from the parameters of Stimulus2 specified in the scenario model (see Fig. 9.5).

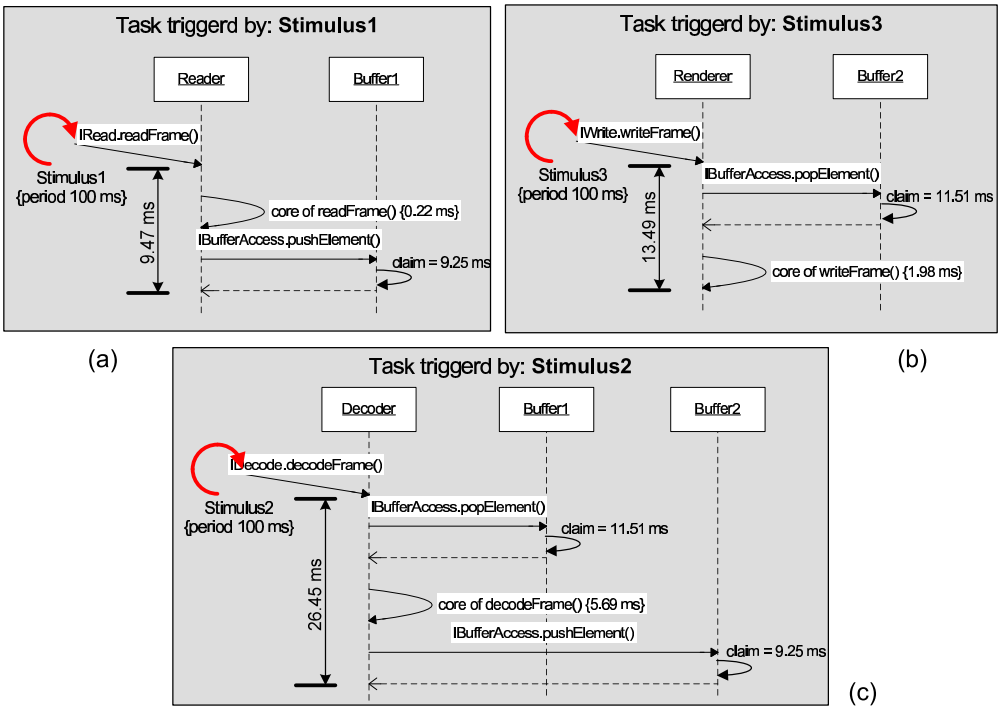

Figure 9.6: *Tasks generated for the scenario: (a) reading task triggered by Stimulus1, (b) decoding task triggered by Stimulus2, (c) rendering task triggered by Stimulus3.*

9.2.5 Scenario Simulation

The execution of the generated tasks is simulated with a rate-monotonic virtual scheduler provided by the CARAT tool. The resulting execution *timeline* is depicted in Fig. 9.7. The timeline indicates all running tasks of the decoder, their predicted *start* and *completion times*, as well as deadlines. The gray boxes in the figure represent the occupation of the CPU caused by their corresponding tasks. The black vertical line in a gray box means a transition from one operation executed to another within the task. The simulation time was set to 1,500 seconds. The figure depicts only the first 400 ms of the simulation execution. From the simulation results, we conclude that the deadlines for all three tasks are met along the whole simulation time (also for non-plotted intervals). The general performance property (processor utilization) was derived from the timeline data. The total CPU utilization is predicted to 49.4%. In the next section, we discuss and compare the predicted results with the real execution data in more detail.

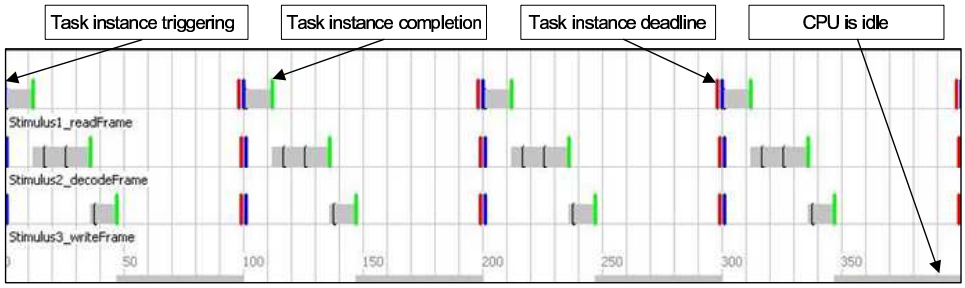

Figure 9.7: *Task execution timeline for the selected scenario, extracted from the CARAT Visualizer tool.*

9.2.6 Experiments and Results on the MPEG-4 Case

After prediction at the design phase, we integrate the software components into an actual MPEG-4 decoder application and execute this on a Linux/32 platform. We aim at obtaining two types of data: (a) timeline data (which task is executed at what moment) and (b) processor utilization data.

First, we compare the acquired (by the Linux Trace Toolkit) timeline data with the predicted execution timeline. The analysis shows a slight difference in the predicted-*vs.*-real execution moments of the tasks. This difference is explained by numerous OS kernel activities (pagefaults, timestamps, i/o calls) interrupting the decoder execution. Thus, the approach does not allow accurate prediction of the starting/completion times of individual tasks. However, the predicted patterns of the task execution coincide well with the real execution patterns. Besides this, the real-time requirements of the tasks (100 ms) are satisfied as predicted by the method. However, this prediction accuracy is obtained under favorable conditions (CPU slack of 50% available). In case of higher decoder CPU utilization, we expect that the real-time requirements would be jeopardized by the kernel activities. It is clear that for those high-load condition scenarios, the OS kernel activities should be also modeled and incorporated into the simulation.

Second, we analyze the accuracy of the predicted performance properties (see Table 9.1). The prediction accuracy error on the average processor utilization appears to be about 10%, and proves to be reasonably stable. The accuracy error on the *time-detailed* processor utilization (granularity = 1 s) varies within 30%. The reason for the strongly varying processor usage is the variable number of macroblocks for decoding of arbitrary-sized video objects (as we mentioned above, the CPU *claims* of each operation are specified for a worst-case scenario). This observation endorses us to enhance the scenario-based method later with *parameter-dependent* modeling.

Finally, we have discovered an interesting phenomenon in the decoder tim-

Table 9.1: *The predicted vs. real execution data comparison of the MPEG-4 decoding application.*

Type of measurements	Predicted exec. data	Real data	Prediction tolerance
Average CPU utilization	49.4%	45.1%	8.7%
Reading Task av. utilization	9.5%	8.7%	8.4%
Decoding Task av. utilization	26.4%	23.6%	10.6%
Rendering Task av. utilization	13.5%	12.8%	5.2%
Time-detailed CPU util. (max)	49.4%	49.1%	0.6%

ing behaviour under memory-overload conditions. In the Linux OS, a concept of virtual memory is used for dealing with memory overload. This virtual memory is implemented by *page swapping*, i.e. releasing memory slots by storing memory data on a disk. Returning this data to the memory upon request takes a relatively long time. In the MPEG-4 decoder, the swapping causes serious latency in the processing (1-2 video frame periods), because the data cannot be read instantaneously. The resulting behaviour is that latency requirements are not met, while the CPU usage is very low. The conclusion is that in the ideal case, the OS system activities need to be analyzed jointly to identify the aspects that hamper accurate predictable system operation. However, modeling the activities caused by a non-real-time operating system is hard and sometimes even impossible due to unpredictability of the OS behaviour. If a real-time OS is used, modeling of the OS behaviour can be performed in the scenario model. The addition of triggers will emulate firing of OS events, while the addition of a service representing an OS activity, will emulate the load of the OS onto the hardware resources.

9.2.7 Conclusion on the MPEG-4 Case

We have exploited a scenario simulation approach that enables prediction of real-time properties of an advanced software-intensive MPEG-4 coding system. The average real-time behaviour of the MPEG-4 decoder can be predicted with high accuracy (within 90%). The prediction accuracy error on the time-detailed processor utilization varies within 30%. As an extra benefit, the timing results give detailed performance information at the design phase. For obtaining the time-detailed accuracy, we have found that integration of a timing model for system activities into our modeling technique is indispensable, because our experiments reveal a large variation of starting and completion times.

The knowledge about the generic computational costs resulting from our

approach, provides important guidelines for efficient software/hardware co-design of multimedia coding systems. The case study shows that the processor, memory and bus usage need to be analyzed jointly, instead of processor analysis only. A second important result is that input parameter dependencies should be incorporated in the method. Third, the system-level activities cannot be neglected during the modeling phase. These three observations have been taken into account in the next iteration on the DeepCompass framework development. This new version of the framework is validated in the following case study on Car Radio Navigation system.

9.3 Car Radio Navigation (CRN) System

In the the second framework iteration, we have extended the DeepCompass framework with the following aspects.

- Facilities are added for modeling and analysis of systems mapped on heterogeneous hardware platforms with multiple processor and network nodes.
- Functionality for analysis of a *robustness* quality attribute. This enriched our set of quality attributes for trade-off analysis and multi-objective optimization.
- Feature for comparing several architectural alternatives by the Pareto-frontiers technique.

The aforementioned framework extensions for resolving performance design trade-offs have been validated by a subsequent case study on a Car Radio Navigation (CRN) system. As a basis for the study, we have taken the benchmark CRN application [101] used also in [102, 47].

We have identified the following goals and constraints for the study. The CRN system has to be constructed according to the component-based paradigm on a cost-limited (at the start not yet defined) hardware platform. The major challenge is to find at an early design stage an optimal system architecture in terms of the vital QAs like latency, robustness and cost. Technically speaking, the objective is as follows: *given* a set of functional and extra-functional requirements as well as a set of software and hardware components, our aim is to *determine* a set of architecture solutions that are optimal with respect to the above-mentioned quality attributes.

Requirements. We divide the requirements into two categories: functional (Fn) and extra-functional (RTn). The main requirements are summarized below.

- *F1:* The system shall be able to gradually (scale = 32 grades) change the sound volume.
- *RT1:* The response time of the operation F1 is less than 200 ms (per grade).
- *F2:* The system shall be able to find and retrieve an address specified by the user.
- *RT2:* The response time of the operation F2 is less than 200 ms.
- *F3:* The system should be able to receive and handle Traffic-Message-Channel (TMC) messages.
- *RT3:* The response time of the operation F3 for one message is less than 350 ms.

Functional decomposition. Requirements analysis lead us to a conceptual software view, as depicted in Fig. 9.8.

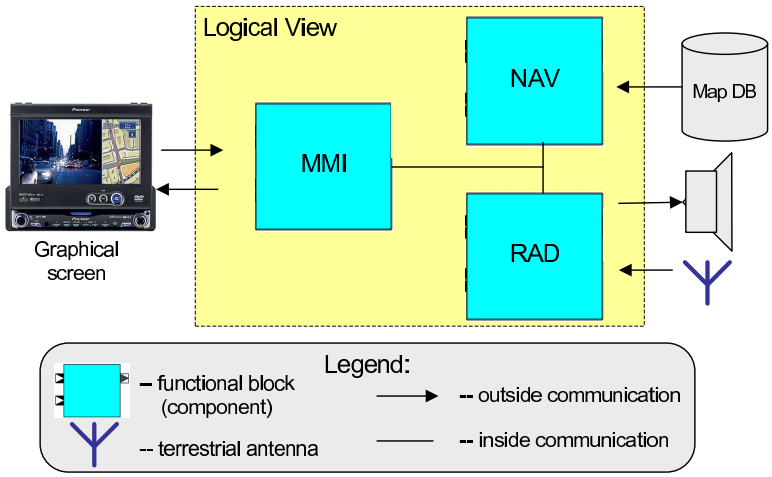

Figure 9.8: *Overview of the CRN system functionality.*

The CRN benchmarking application [101] contains three major functional blocks.

- The Man-Machine Interface (MMI), that takes care of all interactions with the end-user, such as handling key inputs and graphical display output.
- The Navigation functionality (NAV) is responsible for destination entry, route planning and turn-by-turn route guidance giving the driver visual advices. The navigation functionality relies on the availability of a map database and positioning information.
- The Radio functionality (RAD) is responsible for tuner and volume control as well as handling of TMC information services.

In the next section, we illustrate how the DeepCompass framework enables architecture comparison and resolves design trade-offs with respect to multiple performance attributes and cost.

9.3.1 Quest for an Optimal CRN Architecture

For this case study, we implement three ROBOCOP services: RAD, MMI and NAV, which correspond to the above-mentioned CRN functional blocks. These three services and their provides/requires interfaces and operations are depicted in Fig. 9.9(a).

This paragraph explains the interfaces in more detail. The MMI service provides an `IGUIControl` interface and requires to be bound to `IParameters` and `IDatabase` interfaces. The `IGUIControl` interface provides access to three implemented operations: `setVolume()` (handles the volume rotary button request from the user), `setAddress()` (handles the address keyboard request from the user) and `updateScreen()` (updates the GUI display). The NAV service provides `IDatabase`, `ITMC` interfaces and requires operations from the `IGUIControl` interface. The `IDatabase` interface gives access to the `addressLookup()` operation, which queries the address in the database and finds a path to this address. The `ITMC` interface provides access to the `decodeTMC()` operation. The RAD service provides `IParameters`, `IReceiver` interfaces and requires an `ITMC` interface. The two operations implemented by this component are `adjustVolume()` and `receiveTMC()`.

Each service is accompanied by *resource*, *behaviour* (see Fig. 9.9(b), and *cost* models. The operations' resource usage (CPU claim) of a RISC processor is obtained from the CRN benchmark case study [101]. The operation behaviour data is generated from the component source code. For example, the RAD behaviour model describes that the operation `adjustVolume()` synchronously calls once the `IGUIControl.updateScreen()` operation. This model also shows the bus usage of the `adjustVolume()` operation: 4 Bytes. That means the operation sends 4 Bytes of data outside (as an argument of `updateScreen()`).

9.3.2 Defining Architecture Alternatives

We compose a component assembly (see Fig. 9.10(a)) from the available services. We are able to design only one software architecture alternative due to the constraints from the CRN benchmark case study. The three services are instantiated and bound together via pairs of their provided/required interfaces. This assembly satisfies the three defined functional requirements: F1, F2 and F3.

The next phase is to define a set of hardware architectures and map the software components onto hardware. We reuse five feasible alternative hardware architectures with different mapping schemes, as proposed in [101] (see

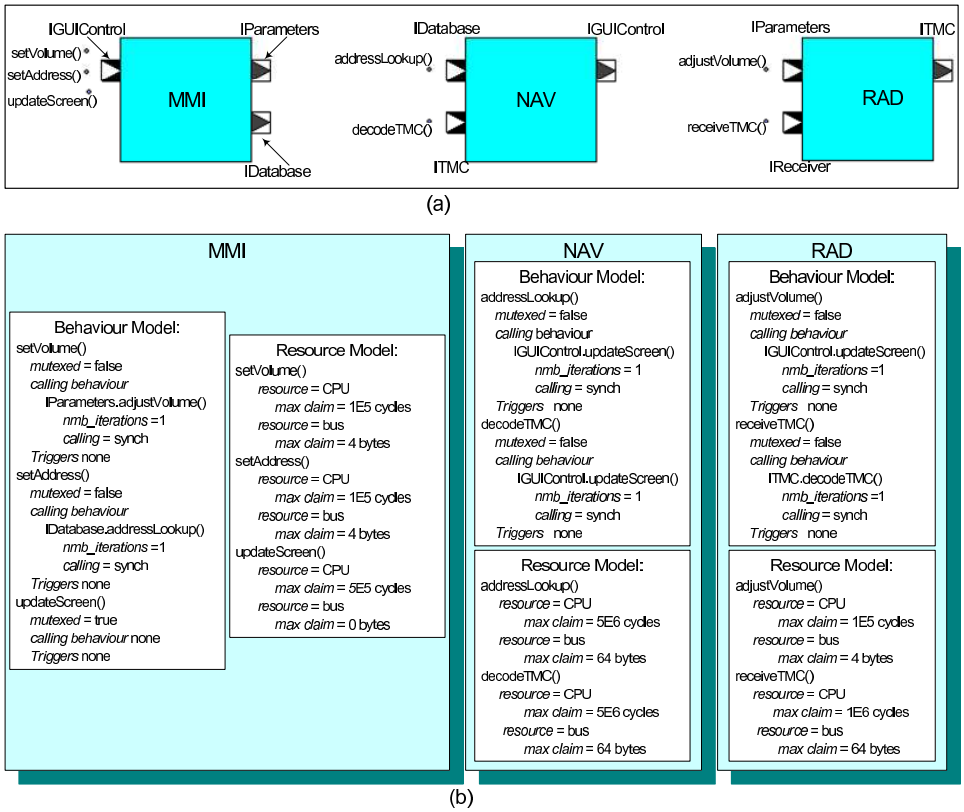

Figure 9.9: *(a) Services used for the case study; (b) behaviour and resource models of the selected services.*

Fig. 9.10(b)). For instance in Architecture A, there are three processing nodes connected with a single bus of 72 kb/s bandwidth. The MMI_Inst service instance is executed (mapped) on a 22-MIPS processor, the NAV_Inst service instance is mapped on a 113-MIPS processor, and RAD_Inst service instance executes on an 11-MIPS processor. The capacity of the processing nodes and communication infrastructure is taken from the datasheets of several commercially available automotive CPUs. The performance analysis and multi-objective optimization is performed for these five solutions.

9.3.3 Scenarios and Task Generation

From the CRN benchmarking case study, we select three distinctive execution scenarios to assess the architecture against the six defined requirements. These scenarios should impose the highest possible load on the hardware resources

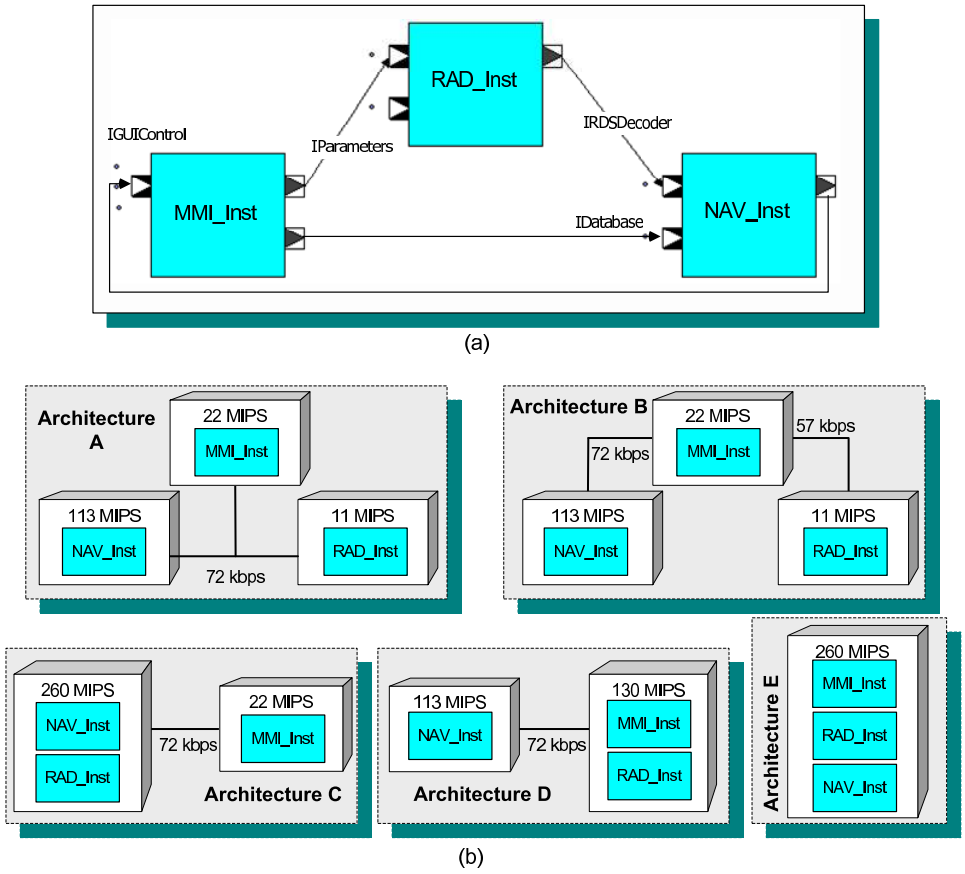

Figure 9.10: *(a) Software component assembly of the CRN system; (b) five alternative architectures for exploration.*

for accurate evaluation of the real-time requirements RT1, RT2 and RT3.

"Change Volume" scenario. The user turns the rotary button and expects instantaneous audible and visual feedback from the system. The maximum rotation speed of the button is 1 second from the lowest to the highest position. For emulating this user activity, we introduce a VolumeStimulus task trigger, which initiates execution of the IGUIControl.setVolume() operation. The trigger parameters are defined in the following way: the event period is set to 1/32 sec, as the volume button scale contains 32 grades. The task deadline is set to 200 ms, according to R1. The trigger and component assembly resemble a scenario model.

For this scenario, the CARAT tool generates (from the behaviour models of participating services) the message sequence chart (MSC) of operation calls

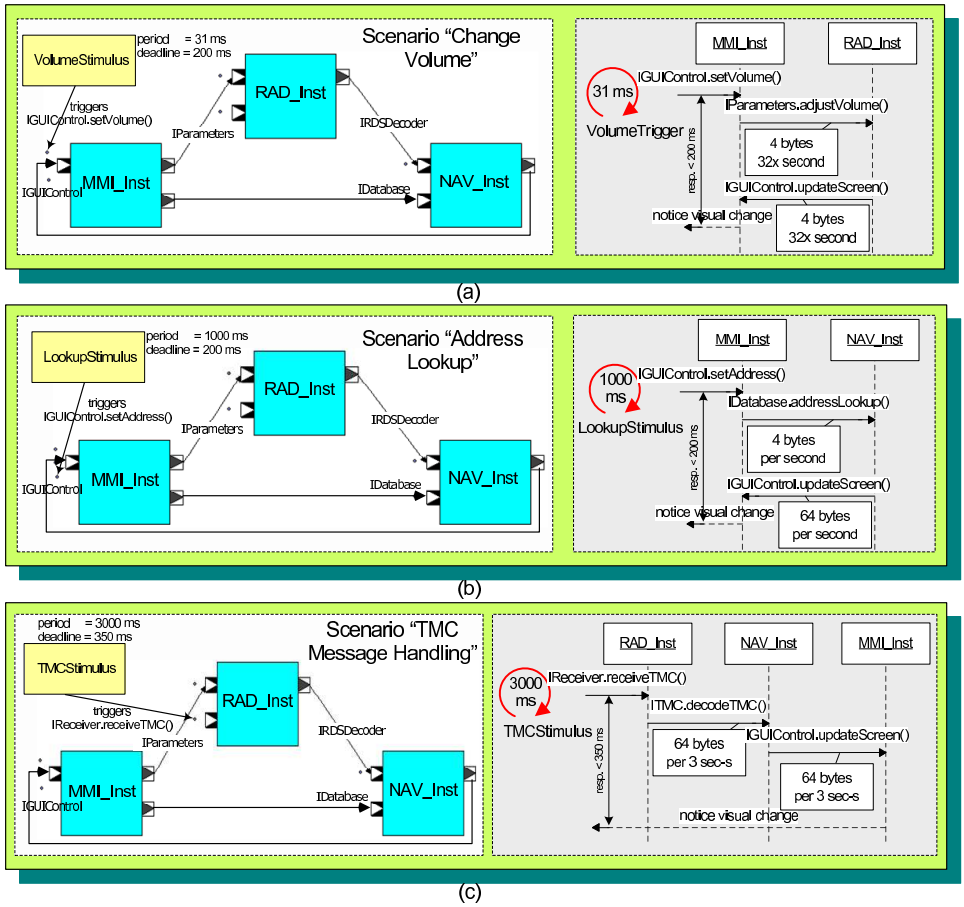

Figure 9.11: *Model and message sequence chart for scenarios: (a) Change Volume; (b) Address Lookup, and (c) TMC Message Handling.*

involved in the task execution. The scenario model and obtained MSC are shown in Fig. 9.11(a). The task is executed periodically (31 ms) and passes through MMI_Inst and RAD_Inst.

"Address Lookup" scenario. A destination entry is supported by a smart *typewriter* style interface. The display shows the alphabet and the user selects the first letter of a street. By turning the letter-selection knob, the user can move from letter to letter; by pressing the knob, the user selects the currently highlighted letter. The map database is searched for each letter that is selected and so on. We assume that the worst-case rate of the letter selection is 1 time per second. This user activity is emulated with a LookupStimulus trigger, which initiates execution of the `IGUIControl.setAddress()` operation.

The trigger period is set to 1000 ms. The deadline for the address lookup task is 200 ms, according to RT2.

The CARAT task-generation procedure outputs the task MSC for this scenario. The obtained scenario model and MSC are shown in Fig. 9.11(b). The task is executed periodically (1000 ms) and passes the MMI_Inst and NAV_Inst service instances.

"TMC Message Handling" scenario. RDS TMC is digital traffic information that enables automatic replanning of the route in case of a traffic jam. Traffic messages are received by the RAD component (in the worst case 1 time per 3 seconds). We introduce a TMCStimulus trigger emulating these TMC messages. The trigger initiates execution of the `IReceiver.receiveTMC()` operation. The period is set to 3000 ms. The deadline for the TMC handling task is set to 350 ms, according to RT3.

The CARAT task-generation procedure results in the task MSC for this scenario. The obtained scenario model and task are represented in Fig. 9.11(c). The task is executed periodically (3000 ms) and passes through three service instances: RAD_Inst, MMI_Inst and NAV_Inst. The fully decoded messages are forwarded to the user.

9.3.4 Simulation and QA predictions

The scenarios sketched above have an interesting property: they can occur in parallel. TMC messages must be processed while the user changes the volume or enters a destination address at the same time. Therefore, we combine these three scenarios into two, in order to obtain a worst-case load on the system resources during simulation. We define *ScenarioA* as a combination of the SetVolume and TMCHandling scenarios, and *ScenarioB* as a combination of the AddressLookup and TMCHandling scenarios. From the processing point of view, both new scenarios execute two tasks in parallel.

Following the DeepCompass process, we simulate the execution of these two critical scenarios for each of the five system architectures. Prior to simulation, the preprocessing of the computation and communication time data is performed as follows. For each processing node, the execution times of all operations to be executed on the node are calculated from the *component resource* and *node performance* models (execution_time = *CPU_claim*_value × processor_speed). The communication time of the operation calls made through the processor boundaries is calculated by dividing the *bus-claim* value of an operation on a bus-bandwidth value, which is already defined in a bus performance model.

The scenario simulation is conducted using a preemptive rate-monotonic algorithm (other policies can also be used). This results in (a) predicted system timing behaviour description and (b) resource consumption of a system for each scenario and task worst-case latencies. First, we analyze the predicted

Table 9.2: *Experimental data of the predicted quality attributes for five architecture alternatives.*

Attribute	Arch. A	Arch. B	Arch. C	Arch. D	Arch. E
Max. task latency against RT1 (RT1=200ms)	37.55 ms	37.55 ms	30.52 ms	9.18 ms	3.58 ms
Max. task latency against RT2 (RT2=200ms)	86.51 ms	86.51 ms	61.49 ms	63.79 ms	21.05 ms
Max. task latency against RT3 (RT3=350ms)	325.05 ms	395.05 ms	101.71 ms	114.12 ms	46.02 ms
Perf. sensitivity (latency increase for TMC handling)	57.6%	51.1%	3.2%	3.1%	0.0%
Cost, euro	290	305	380	335	340

task latencies against the real-time requirements RT1, RT2 and RT3 for each of the five architectures (see Table 9.2).

Analyzing the table data, we conclude that except for Architecture B, the remaining four architectures satisfy the given real-time requirements. Architecture B does not satisfy the requirement RT3, because it has a TMCHandling task latency that is higher than 350 ms. Architecture A can be considered fast enough, while Architecture E gives the fastest solution. Then, we analyze the architecture *robustness* as a performance *sensitivity* to the changes in the input event rates (arrival period of the three stimuli). We increase the data rate of the three stimuli by 5% (i.e. VolumeStimulus to 33.6 events/s, LookupStimulus to 1.05 events/s and TMCStimulus to 0.35 events/s). Afterwards, we re-simulate the adjusted scenarios and obtain new task latencies. The fourth row in Table 9.2 describes the increase of the latency of the TMC handling task as a percentage of the normal latency per architecture. For instance, the end-to-end delay of the TMC message handling task for Architecture A increases by 57.6%! This occurs due to a high overload of the 22-MIPS processor in this scenario.

The system-*cost* attribute is calculated as a cumulative *cost* of the system hardware and software components. The software-component cost is defined with correlation to the component source-code complexity (in reality, the cost

of a third-party component is defined by the component producer). The cost of the hardware components is calculated from the available market prices. The total calculated cost for each architecture is given in Table 9.2. The most expensive architecture is number C due to the costly high-performance processing nodes.

9.3.5 Analysis of Architecture Alternatives

The task latencies, robustness and cost attributes are selected as main objectives for our design space exploration. Using the CARAT Pareto analysis tool, we obtain several two-dimensional Pareto graphs. Two of them, *robustness vs. cost* and *performance vs. cost* are depicted in Fig. 9.12.

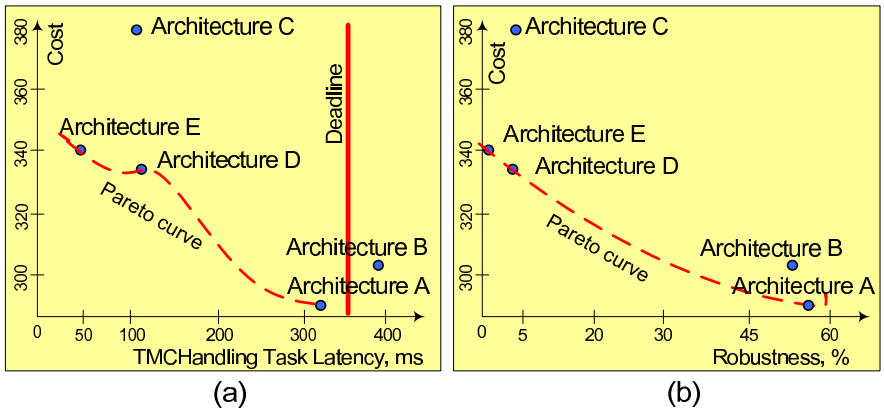

Figure 9.12: *Pareto exploration graphs based on (a) performance vs. cost, (b) robustness vs. cost*

The graphs are evaluated as follows. The Pareto curve is drawn by connecting the alternatives that are closest to the origin. This curve defines a set of optimal alternatives. With respect to the cost-robustness trade-off (see Fig. 9.12(b)), the optimal architectures are E, D and A, because they create the curve closest to the null-coordinate point. The alternatives C and B are non-optimal. When confining to one of those three optimal alternatives, a final choice depends on a weighting function (priority) for the cost and robustness attributes. If cost has a higher priority, then Architecture A should be selected. If performance sensitivity is a critical factor, then Architecture A is not the best candidate. Moreover, looking at the cost-performance trade-off (see Fig. 9.12(a)), we can observe that TMC task latency for Architecture A is close to its deadline. Therefore, a low robustness (57.6%) of Architecture A cannot be tolerated.

With respect to the cost-performance trade-off, again the optimal alternatives are E, D and A, though C is not positioned on the hypothetical ideal Pareto curve. Architecture B is not competitive because its TMC task latency is higher than the task deadline. Although it has a low cost, Architecture A provides a low performance and insufficient robustness, and is therefore also omitted.

Concluding, the Architectures E and D are considered to be the optimal alternatives. If the cost weighting function is higher than the performance or robustness weighting function, Architecture E can be adopted for further development and *vice versa*. In addition, we may also re-iterate the DeepCompass process to achieve acceptable performance for the less costly Architecture A. For instance, we can add a new software component TMCHandler, which reduces the TMCHandling task latency, or re-dimension one of the processing nodes. Another optimization technique would be to reduce the cost of Architecture E, by sacrificing (within acceptable range) its performance and robustness.

9.4 JPEG Decoder Application

After having evaluated the second case study (CRN), we have performed a third refinement on the DeepCompass framework. In this refinement, we have added facilities for design and analysis of multimedia applications. Most multimedia systems are built using the pipes-and-filters architectural style. In this style, active software components are frequently used. Active components start and run their own process (thread of control), which reads data from a buffer, processes the data and stores the processed data into a buffer. Summarizing, we have extended the framework with the following facilities for modeling and behaviour analysis of multimedia systems.

- Process component model for specification of active components.
- Virtual services representing memory buffers.

In this section, we present the design case study performed on a JPEG multimedia application. The purpose of the case study is to validate the framework for relatively complex industrial applications. The JPEG decoder software is adopted from the JPEG Decoding Network (JDN), as developed in [67].

9.4.1 Services Identification

We have identified the main software building blocks of the JDN system and, for each block, we have developed a ROBOCOP service. The JDN system has been implemented as a process network using YAPI (similar to FIFO)

buffers [66]. Therefore, we have made services of two types: executable services and virtual buffer services (see Fig. 9.13). Executable services are conventional services that have provides and requires interfaces and perform some computations, while the virtual buffer services represent YAPI buffers that only provide a communication infrastructure. The executable services communicate with each other via virtual buffer services. The modeling specification for a virtual buffer service is given in Section 5.3.

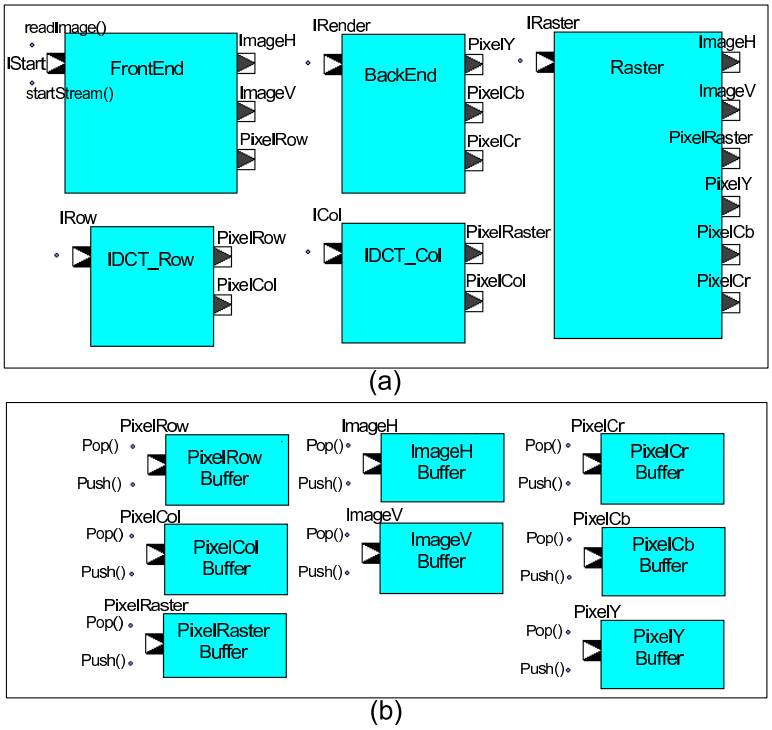

Figure 9.13: *JPEG software services: (a) executable services, (b) virtual buffer-services*

Fig. 9.13(a) depicts the following executable services. The FrontEnd service provides the `IStart` interface that offers start-stream functionality, and requires the following three interfaces: `ImageH`, `ImageV`, `PixelRow`. These interfaces are used by the service to send three de-multiplexed and quantized streams of pixels for further processing. The IDCT_Row service implements one-dimensional Inverse Discrete Cosine Transformation (IDCT) and requires two interfaces to be bound. The `PixelRow` interface is used to acquire a row of pixels from the streamed 8×8 pixel block. The `PixelCol` interface is used to send the partly processed pixel block further on. The IDCT_Col service also provides functionality of the one-dimensional IDCT and requires two in-

terfaces. The `PixelCol` interface is used to acquire a column of pixels from the streamed 8×8 pixel block. The `PixelRaster` interface is used to send the processed pixel block for a rastering operation.

The Raster service contains original downscaling, rastering, vertical scaling, image-to-line operations and horizontal scaling. It requires six interfaces to be bound. The `ImageH`, `ImageV` and `PixelRaster` interfaces are used by the service to receive the pre-processed pixel streams for its own processing. The `PixelY`, `PixelCb` and `PixelCr` interfaces are used to send the processed pixel streams for rendering on a screen. The BackEnd service provides rendering functionality. It pulls the pixel streams via its three required interfaces: `PixelY`, `PixelCb` and `PixelCr`.

Fig. 9.13(b) shows virtual buffer services that provide the communication infrastructure. The PixelRowBuffer service provides `PixelRow` interface with two operations: `pop()` and `push()`. The `push()` (`pop()`) operation can be used by other services to write (read) a row of pixels to (from) a buffer. A similar type of functionality is provided by other buffer services (PixelColBuffer, PixelRasterBuffer, ImageHBuffer, PixelHBuffer and PixelY (Cr/Cb) Buffer). The differences between the buffer services refer only to the size of the single element to be written (read) and the buffer capacity.

9.4.2 Specification of Component Models

Following the DeepCompass framework phases, we have specified supplementary models for the previously discussed services. As explained in Section 5.3.2, the behaviour model is used for passive services that do not implement and execute any processes or threads upon their activation. The process model is used for active services that start and execute one or more processes. The data for behaviour and process models have been obtained by the source code analysis of the services. Analyzing the source code of the available executable services, we have identified that the FrontEnd service is passive, while the rest of the executable services are active. The source code analysis leads us to the behaviour- and process- models of the services. The service's stand-alone profiling on a reference RISC processor provides data for the resource models. All models reflect a decoding mode of the JPEG picture in 4:2:0 format with a picture size of 1004×669 pixels. For simplicity, we omit the parameter-dependent specification here.

Fig. 9.14 illustrates a simplified version of the models of the FrontEnd, IDCT_Row and BackEnd components. The next paragraphs explain the models in detail.

For each implemented operation, the behaviour model of the FrontEnd service specifies the sequence of calls made to operations of other interfaces. The model specifies that the implemented operation `startStream()` first calls once the `push()` operation through the `ImageH` and `ImageV` interface (send-

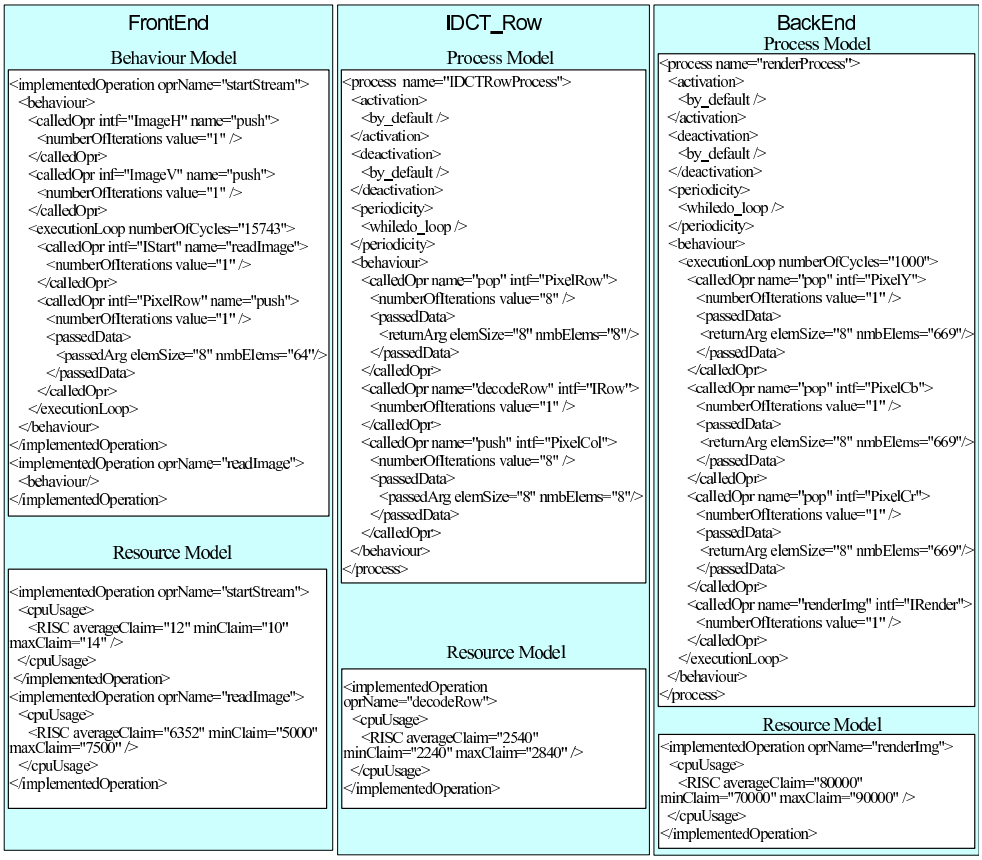

Figure 9.14: *Models of the FrontEnd, IDCT_ Row and BackEnd services.*

ing the height and width parameters of the JPEG picture). Afterwards, our operation enters the loop of 15,743 cycles. In each iteration of the loop, the `startStream()` operation subsequently calls the `readImage()` operation through the `IStart` interface and the `push()` operation through the `PixelRow` interface. The latter call implements functionality of sending the processed data to the buffer service. The volume of the sent (as an operation argument) data is represented in the *passedData* token, which shows that 64 elements of the size of 8 bits are sent to the buffer. The second implemented operation `readImage()` does not call any other interfaces, thereby its *behaviour* token is empty.

The resource model of the FrontEnd service specifies average, minimum and maximum numbers of processing cycles that each implemented operation needs for its execution. For example, the `startStream()` operation claims on the average 12 cycles of a RISC processor (see the token *cpuUsage* in Fig. 9.14),

while the `readImage()` operation takes on the average 6,352 processing cycles.

The process model of the IDCT_Row service (second column in Fig. 9.14) describes a process that the service implements itself. The token *by_default* specifies that the process is initialized and run upon service start-up. The token *whiledo_loop* tells that the implemented functionality within the process is executed within an infinite loop. The *behaviour* token describes the loop functionality in terms of a sequence of calls to operations provided by other interfaces. The `pop()` operation from the `PixelRow` interface is the first called operation. The call is repeated 8 times (see token *numberOfIterations*). By this operation, the process reads data from a buffer providing the `PixelRow` interface. The amount of data to be read is shown in the *passedData* token. The *returnArg* token specifies that 8 elements with a size of 8 bits are read from the buffer during the operation invocation. The next called operation is `decodeRow()` from its own interface `IRow`. The last invocation within the loop is the `push()` operation through the `PixelCol` interface. The operation is called 8 times. The passed arguments (data stored to a buffer) is an array of 8 elements of 8-bit size. The IDCT_Row resource model specifies that the `decodeRow()` operation claims on the average 2,540 cycles of a RISC processor.

The virtual buffer services are not supplemented with models because they do not provide any processing functionality. The initialization of the buffer sizes is to be performed at the software architecting phase.

9.4.3 JPEG Software Architecture

We have designed the software architecture for the JPEG decoder using the CARAT repository and graphical editor tools. We have instantiated the defined services and bound their respective provides and requires interfaces (see Fig. 9.15).

The architecture satisfies the functional requirements of a JPEG decoder, which works as follows. The Front_End service reads the input stream, demultiplexes it and sends the processed data to the ImageHBuffer, ImageVBuffer and PixelRowBuffer services. The IDCT_Row service reads the data from PixelRowBuffer, performs IDCT transformation of rows in the 8×8 pixel blocks and sends the processed data to PixelColBuffer. The IDCT_Col service fetches the data from the PixelColBuffer, performs IDCT transformation of columns in the 8×8 pixel blocks and stores the processed data to PixelRasterBuffer. The Raster service reads data from the ImageHBuffer, ImageVBuffer, PixelRasterBuffer services. After these actions, this service rasterizes and scales the processed image, and sends the YCrCb pixel components to the PixelYBuffer, PixelCrBuffer and PixelCbBuffer services, respectively. The BackEnd service fetches the pixel components from these three buffers, and renders the image on a screen.

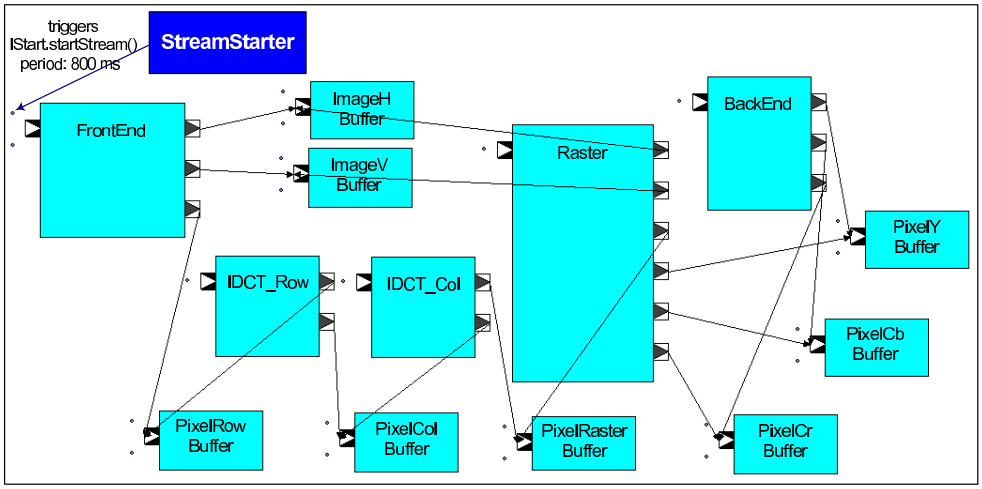

Figure 9.15: *JPEG scenario model consisting of the software architecture and starting-stream trigger.*

Scenario identification

According to the DeepCompass process, the software architecture should be incorporated in the execution scenario model by specifying environmental or user-based events that trigger the execution flows within the system. In the JPEG decoder, such a user-based event is a 'view picture' command from a user that triggers the picture processing functionality. Upon the 'view picture' command, the keyboard listener invokes the `IStart.startStream()` operation of the FrontEnd service. We have defined this trigger using the CARAT software architecture editor (see Fig. 9.15). Besides this, we add a link to the triggered `IStart.startStream()` operation and specify the trigger period and deadline properties as follows. The period is set to 800 ms, due to the expectation that the user's fastest viewing speed (and the 'worst-case' conditions for the system resources) is 10 pictures per 8 seconds. This trigger, together with the component assembly, represents the scenario model. In the scenario model, we set the storage capacity of each virtual buffer service to 1 Mbits.

9.4.4 JPEG Hardware Architecture Alternatives

Using the hardware architecture editor of the CARAT toolkit, we have designed five hardware architectural alternatives which are depicted in Fig. 9.16. Our reasoning behind these architectural choices is as follows. We aim at defining a spectrum of hardware architectures by varying the number of processor (memory) nodes and the type of topology.

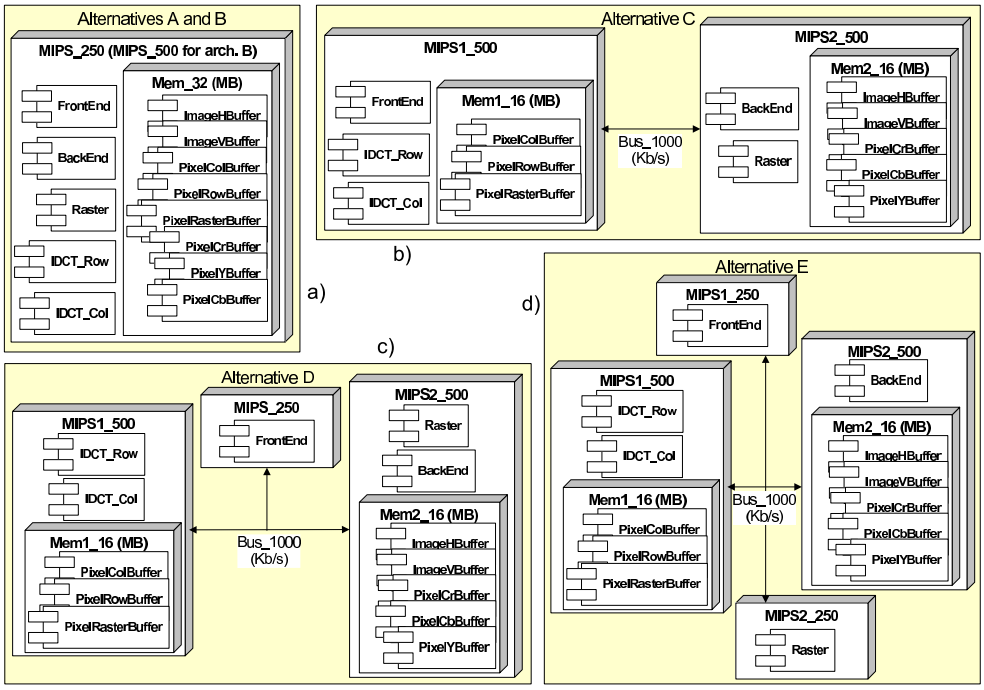

Figure 9.16: *The hardware architecture and mapping of (a) Alternatives A and B, (b) Alternative C, (c) Alternative D, (d) Alternative E.*

The Alternatives A and B, shown in Fig. 9.16(a), have one processor node with a processing speed of 250 and 500 MIPS, respectively, and the local memory with a capacity of 32 MB. The Alternative C (see Fig. 9.16(b)) has two processors with a processing speed of 500 MIPS each and one local memory (16 MB) per processor. The processor communication is realized via a network with a bandwidth of 1 Mbit/s. The communication infrastructure in the next two alternatives is also implemented by this network. Alternative D deploys three heterogeneous processors with different operation speed. One of the processors has no local memory. Alternative E (see Fig. 9.16(d)) defines four heterogeneous processors. Only two of these processors have access to their own local memory of 16 MB.

9.4.5 SW/HW Mapping Alternatives of JPEG Application

Having defined the software and hardware alternatives, we proceed to static mapping of the software services onto the hardware IP blocks. The executable services are mapped on processing nodes, meaning that the code of an individual service will be executed on a predetermined processor. The virtual buffer

services are mapped on memory blocks, implying that the data stored to a buffer will be allocated on a predetermined memory block.

Using the CARAT toolkit, we map the services from the scenario model on each of the five predefined hardware architectures (see Fig. 9.16). The reasoning behind the mapping choices is twofold. Firstly, communicating services should be mapped as close as possible to each other. Secondly, the services imposing a high load on the processing resources (this can be checked from their resource models) should not be mapped on the same resource. For instance, Alternative C defines that the FrontEnd, IDCT_Col and IDCT_Row services are mapped on the MIPS1_500 processor, while the Raster and BackEnd services are executed on the MIPS2_500 processor. The PixelColBuffer, PixelRowBuffer and PixelRasterBuffer services are mapped on the Mem1_16 local memory block. The rest of the virtual buffer services are mapped on the Mem2_16 local memory block.

9.4.6 Synthesis of the Executable System Model

Once the software and hardware architectures are defined and the SW/HW mapping is specified for an architectural alternative, the CARAT toolkit synthesizes the executable system model of this alternative. For the JPEG decoder, the synthesis results in five identified tasks: one triggered by the start-stream event and the other four triggered by the processes running in the IDCT_Row, IDCT_Col, Raster and BackEnd services. The synthesized call-graphs of the tasks are relatively simple. Each of them is propagating only to the neighboring buffers. For instance, the task initiated by the process in the IDCT_Row service has the following call graph extracted from the related component process model (see Fig. 9.14): 8*`PixelRow.pop()`, 1*`IRow.decodeRow()`, 8*`PixelCol.push()`.

The obtained call graphs of tasks are applied to a scenario model and a SW/HW mapping specification. Knowing the operations executing in a task call graph, the CARAT tool scans the SW/HW mapping specification to find the processing nodes on which the services implementing these operations are deployed. It results in a full specification of the task allocations in the system architecture.

These steps result in the executable system model that is employed for performance analysis.

9.4.7 Performance Analysis

We have simulated the five design alternatives of the JPEG decoder using a round-robin scheduler for the processing nodes and a first-come-first-served scheduler for the communication lines. The duration of simulation has been set to 5M microseconds. The input to the performance analysis are the executable

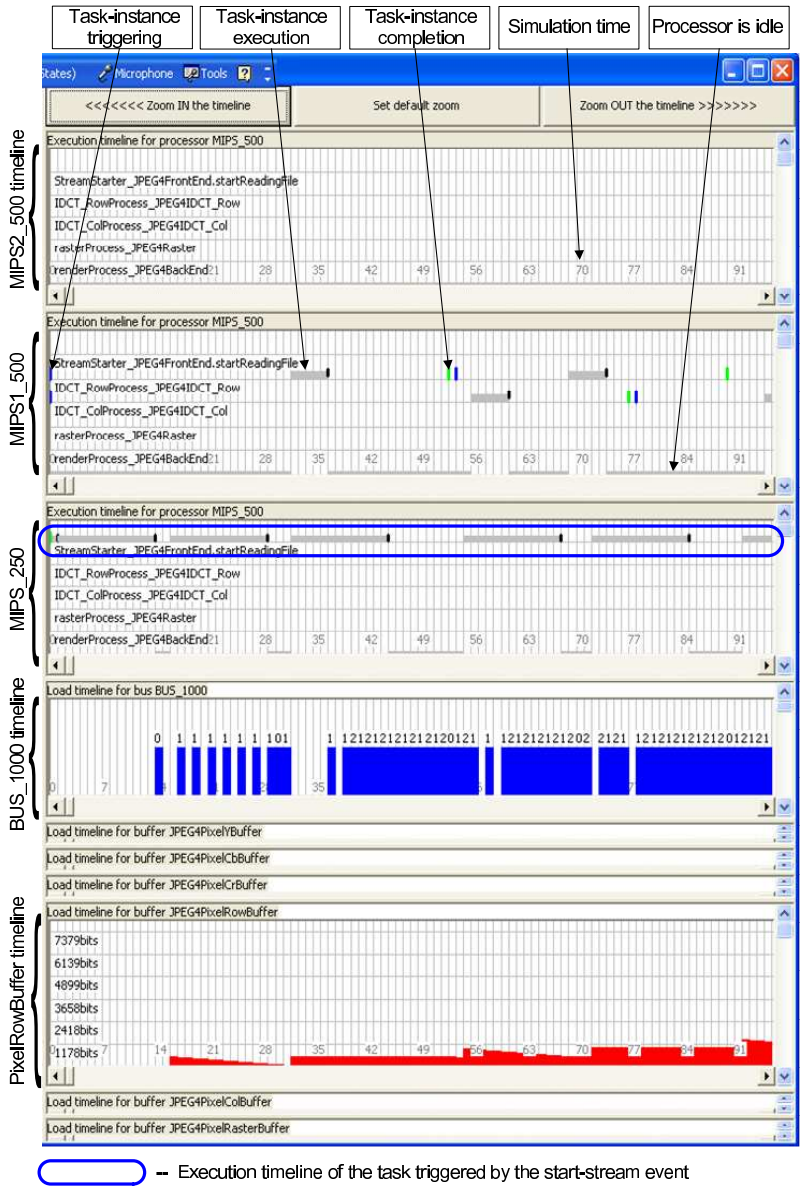

Figure 9.17: *Execution timelines of the Alternative D.*

system models of the alternatives. The analysis results can be divided in two categories: (a) resource usage timeline[2] indicating which task is executed at

[2]Although this name better covers the meaning, in the sequel we rename *resource usage timeline* to *task execution timeline* to preserve analogy with the previous cases.

Table 9.3: *Obtained performance data for five architectural alternatives (Alt.) for the JPEG application.*

Attribute, worst case	**Alt. A**	**Alt. B**	**Alt. C**	**Alt. D**	**Alt. E**
Completion latency (ms)	1412	718	395	343	780
MIPS_250 util., %	98.41	-	-	61.16	-
MIPS_500 util., %	-	97.89	-	-	-
MIPS1_500 util., %	-	-	54.95	29.35	21.42
MIPS2_500 util., %	-	-	59.01	65.09	16.80
MIPS1_250 util., %	-	-	-	-	31.87
MIPS2_250 util., %	-	-	-	-	38.08
Bus_1000 util., %	-	-	25.04	80.76	99.54
Hardware cost (abstract units)	1.2	2.2	4.7	5.7	6.7

every moment in time (see Fig. 9.17), and (b) predicted performance properties of the alternatives, such as utilization of hardware resources, task response time, completion time and number of missed deadlines (see Table 9.3).

The task execution timeline is analyzed to understand the task interleaving on individual hardware resources, identify where and why a task misses its deadline and analyze performance bottlenecks. For each processor timeline, we show the tasks executing the operations from the components mapped on the processor. The bus-load timeline represents the bus utilization imposed by the communicating operations in the tasks. The buffer utilization timeline shows the consumed buffer capacity at each moment in time. For instance, Fig. 9.17 depicts the obtained partial timelines from 0 to 94 microseconds of the hardware nodes of Alternative D. The figure shows the timelines of all three processors used in the system, the bus timeline and the PixelRowBuffer timeline. The timeline analysis shows that at upon the start-stream event, the stream-reading task executes on the MIPS_250 processor and writes the processed data into the PixelRowBuffer via the Bus_1000 network. Once there is enough data in the PixelRowBuffer service, the process implemented by the IDCT_Row service executes on the MIPS1_500 processor and stores the processed data into the PixelColBuffer via the Bus_1000 network also. The MIPS2_500 processor is idle at this time period, because the Raster and BackEnd processes (tasks) assigned onto this processor are waiting for the data to appear in their corresponding input buffers.

The main criterion for the performance evaluation is the completion latency of the processing of one image. The *completion latency* is the time between the arrival of the stream-start event to the FrondEnd service and the moment when the last pixel of the image is rendered on the screen by the BackEnd service process. We have set the real-time requirement that the completion latency should not exceed the predefined period of the start-stream event (800 ms). From the end-user point of view, the requirement implies that the system should allow viewing images with a rate of 10 pictures per 8 seconds.

Table 9.3 aggregates the simulation results of the five architectural alternatives. Except for the completion latency, the table shows the utilization rates of the hardware resources and hardware cost. For simplicity, the hardware cost is calculated in abstract units. The abstract unit is based on normalizing the cost of the resources to the cost of the MIPS_250 processor. The MIPS_250 (MIPS_500) processor cost is set to 1 (2) unit(s), the Mem_32 (Mem_16) costs 0.2 (0.1) units, and the Bus_1000 cost is set to 0.5 units.

9.4.8 Exploiting the Trade-Offs

For these five alternatives, we perform a trade-off analysis with respect to the latency and hardware cost attributes. The CARAT tool converts the latency and cost values shown in Table 9.3 into a Pareto graph (see Fig. 9.18). The Pareto graph positions the five architectural alternatives in the two-dimensional space, according to the obtained values for completion latency and normalized hardware cost.

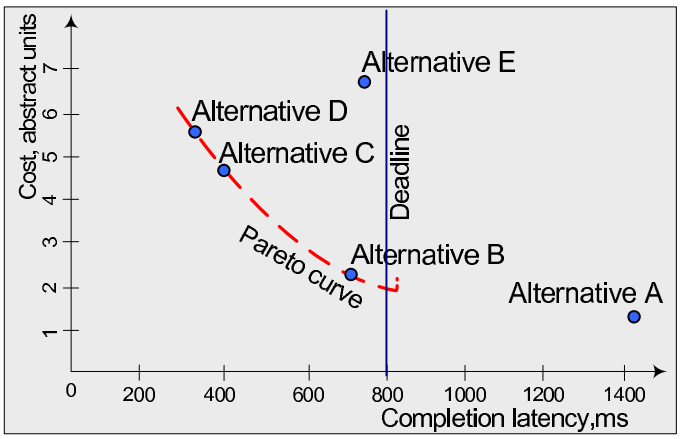

Figure 9.18: *Pareto analysis with respect to hardware cost and completion latency attributes.*

Our resulting Pareto-curve passes through Alternatives B, C and D, which in our case, are optimal solutions. Alternative E is not optimal due to its high

cost and completion latency. Alternative A is not further considered because its completion latency is higher than the specified deadline (800 ms).

A further selection of the architecture alternatives depends on the priorities among the quality attributes. If the cost has higher priority, then Alternative B should be selected. If completion latency is a critical factor, then Alternative B is not the best candidate. Moreover, the latency of Alternative B is close to its deadline. The most balanced solution is Alternative C that combines reasonable cost with high performance.

Analysis of the resource utilization data in Table 9.3 helps to identify performance bottlenecks and optimize these architectures. For example, the high bus utilization (80%) in Alternative D imposes relatively high latency, while its three processors are not fully loaded. Increasing the bus bandwidth will lead to excellent performance at reasonable cost. Another optimization technique is to increase the clock frequency of the single processor in Alternative B. This leads to sufficient performance with low hardware cost.

9.5 Conclusions on Case Studies

In order to validate the presented DeepCompass framework and the scenario-based method, we have performed design and analysis case studies on three real-time applications: an MPEG-4 Decoder, a CRN System and a JPEG Application. Besides the validation of the performance prediction techniques, we have additionally applied the architecture optimization method to the CRN and JPEG systems. After the MPEG-4 Decoder case, we have extended the framework with (a) parameter-dependent modeling, (b) support for heterogeneous processor platforms and (c) the architecture optimization method. The parameter-dependent modeling has proven to be important to cope with large variations in the input-dependent computations. The extension towards heterogeneous platforms is indispensable because this hardware setup is state-of-the-art in the embedded systems domain. The architecture optimization method is needed to enable systematic and balanced improvements starting from the initial architecture. The second case study (CRN) directly led to incorporating the facilities for the pipes-and-filters architectural style, which features sequences of stream-oriented processing with high throughput. This architecture style is broadly applied in the design of multimedia processing systems.

With respect to the *accuracy of performance predictions*, the case studies have revealed the following results. The MPEG-4 decoder showed 30% deviations from the profiled measurements for worst-case task latencies and 10% deviations for processor utilization. The CRN system showed deviations within 5% (with a slight shift to more optimistic worst-case task latencies) in comparison to other well-established performance prediction approaches [95] [48]. The

Table 9.4: *Efforts (in working hours) spent for the three case studies.*

Case study / Activity	Modeling Components	SW/HW Architect. Modeling	Synthesis Simulation	Perform., Trade-Off Analysis	Total
1. MPEG-4	12	9	1	4	26
2. CRN	10	13	3	8	34
3. JPEG	20	14	4	10	48

JPEG application has shown 15% deviations from the profiled measurements of the realized application for worst-case task latencies and 5% deviations for processor, bus and memory utilization.

At this point, the reader may be interested in the effort that is involved performing the architecture analysis and optimization for these three case studies when using the DeepCompass framework. We have measured this effort calculating the time periods for each activity. For each case study, these activities include: component modeling, SW/HW architecture modeling, synthesis and simulation, performance analysis and trade-off analysis. Table 9.4 summarizes the recorded efforts (in working hours) spent for each activity. The average time spent for a case study equals 36 hours, which is equivalent to 4.5 working days.

With the achieved extensions and refinements resulting from the case studies, the final framework is applicable to a broad spectrum of *architectural styles*, *communication styles* and arbitrary *hardware topologies*. In these case-studies we have used (a) pipes-and-filters and blackboard architectural styles; (b) synchronous, message-based and buffers-based communication styles; and (c) hardware topologies with homo- or heterogeneous processing nodes, single and multiple processors, as well as different types of memory blocks.

Chapter 10

Conclusions

10.1 Conclusions of the Thesis

Performance properties of real-time systems are vital and, at the same time, most challenging properties to predict, enforce and measure. Performance depends on multiple dynamic factors related to software, hardware and user / environment behaviour. Performance failures cause project delays, cost overruns, and even abandonment of projects. In order to avoid these failures, the performance properties should be predicted and analyzed already at the early design phases of a project. Prediction of performance properties at an early phase is facilitated by using a 'predictable assembly' of components, as introduced by the CBSE community. In a predictable assembly, the functional and extra-functional properties of individual components are used to reason about the functional and extra-functional properties of a component composition. Unfortunately, an integrated solution for performance-centric predictable assembly with the following combination of characteristics is missing. First, a solution enabling an automated synthesis of system performance properties from the related component properties; second, it requires low modeling and analysis effort; third, a solution provides a reasonably high prediction accuracy.

This thesis presents a new framework called *DeepCompass* (see Chapter 4), which features the three aforementioned characteristics. The DeepCompass framework defines a design process which guides an architect through an iterative design cycle, while focusing on performance properties. The iterative cycle contains the following phases. Firstly, rapid construction of a number

of alternative architectures from the available software and hardware components. Secondly, for each alternative, synthesizing the models of these individual components into a system-wide performance-related model. Thirdly, analyzing these performance models with respect to bottlenecks and the aspects where the alternative does not satisfy the requirements. These three phases save effort and help in selecting promising alternatives and eliciting the best directions for their optimization. Fourthly and finally, the iteration cycle concludes with a performance optimization of promising architectures, while keeping other properties (e.g reliability, cost of materials) within defined boundaries. Depending on the outcome of this phase, an architect may repeat the iteration cycle (without the construction phase), or select the best architecture alternative for implementation.

From a *design-process view*, the DeepCompass framework features the following principal benefits.

- *No system implementation is needed* within the iteration cycle. The design and analysis can be performed *prior to the purchase of the components*, because the system analysis is based purely on component models. An architect needs to obtain (supposedly free of charge) only the corresponding models of the components, design a system and conduct the performance analysis.
- *Rapid architecture prototyping and performance analysis* is enabled. This is achieved by combining both scenario-based and model-based principles. Synthesis of component models allows to characterize system behaviour in a consistent and automated way, while usage of scenarios allows an architect to avoid analysis of all possible system states and focus only on the critical operational modes.
- The performance analysis can be carried out for systems constructed from an *arbitrary set of third-party software and hardware components*. This generic feature is achieved by applying the CBSE principles and providing a rich model set for various types of components. An assumption is that each individual software or hardware component has to be accompanied with pre-defined models specifying the internal properties of that component.

The DeepCompass framework is based on a scenario-based analysis method as described in Chapter 5. One cornerstone of the scenario-based method is the use of *composable models* of individual software components and hardware blocks. The models are specified at the component-development time and are shipped in a component package. At the system design, thus component-composition phase, the models of the constituent components are *automatically synthesized* into an executable system model. The executable system model represents the dynamic characteristics of a system and contains the detailed

data about the tasks running in a system. Simulation of this model provides performance property values.

Another cornerstone of the method is the usage of *critical scenarios*. In this thesis, we advocate considering only critical scenarios for performance analysis. The scenario-based approach helps an architect to focus only on those scenarios that may potentially hinder performance requirements, which saves development time and effort. This scenario-based method allows the analysis of complex systems having a large number of states and configurations.

The scenario-based method features the following benefits.

- The method enables *accurate performance predictions* in addition to rapid prototyping and analysis. The accuracy of performance predictions is computed as the deviation between the predicted and actual worst-case values of performance properties. The case studies described in Chapter 9 show deviations within 30%, obtained in a short time period. The average time spent on modeling and analysis of multiple alternatives for one iteration cycle proved to be only 36 working hours.
- The method supports *all common architectural styles*, e.g. pipes-and-filters, blackboard and client-server, and can be used for systems having an *arbitrary hardware topology* with homo- or heterogeneous processing nodes.
- The method enables modeling of (a) *parameter-dependent* behaviour and hardware resource usage of software components, and (b) *complex behaviour patterns* of components, such as loops and conditional statements.
- The method requires *low modeling and computational effort*. This is achieved by accurate selection of modeling primitives and a relatively high abstraction level of individual component models.

However, this scenario-based method imposes a number of drawbacks. The method requires some experience in identifying the correct critical scenarios and does not guarantee completeness of the selected set of critical scenarios. To address this shortcoming, we have interviewed experienced architects from different industrial domains. The interviews revealed (see Chapter 8) that in practice, architects do use critical scenarios in analysis and never explore the complete state space of a system. For our purpose, we have accumulated the reported experience into a set of guidelines for the identification of correct critical scenarios and their usage in practice.

In Chapter 6, we have performed a comparative review of the architecture optimization approaches that resulted in an abstract optimization method. This abstract method defines the following key steps in optimization: QA analysis, design space analysis, identification of optimization guidelines, and generation of new alternatives. In our opinion, this optimization cycle can be applied to a broad set of optimization approaches. We have found that

the DeepCompass framework can be mapped onto the steps of this abstract method. The automation of all the steps was only partially realized in DeepCompass, in particular, the QA analysis and design space analysis steps. The completion aspects of the automated optimization are discussed in the last section of this chapter.

The *CARAT software toolkit*, presented in Chapter 7, supports an architect in all DeepCompass framework phases, starting from component selection from the repository and ending with code generation for an application. The toolkit enables smooth transitions through design phases by keeping all involved data at a single location and by providing seamless communication between its modules. Additional objecties of the toolkit are to automate complex computational operations and to provide graphical means for designing the alternatives and visualizing the analysis results.

In order to validate the presented DeepCompass framework and the scenario-based method, we have performed *case studies on three real-time applications*: an MPEG-4 Decoder, a Car Radio Navigation (CRN) System and a JPEG Application. These case studies contain all development phases, starting from the requirements elicitation and ending with the profiling of the deployed system on the actual device. Besides the validation of the performance prediction techniques, we have applied the architecture optimization method to the CRN and JPEG systems.

The MPEG-4 and CRN case studies have revealed that the framework initially lacked facilities for analysis of systems that have parameter-dependent behaviour and that are mapped onto an arbitrary multi-processor hardware topology. Moreover, these case studies have shown the need for support of multimedia systems designed in the pipes-and-filters architectural style. Using this feedback, we have enriched the initial framework with facilities for parameter-dependent modeling, process models for active components and multi-processor simulation. All these facilities have been validated by the third JPEG case study. However, the majority of the framework functionality was validated by all three case studies. The conclusions on the obtained prediction accuracy and effort required by the framework are discussed in the following section.

10.2 Discussion on Research Questions

Let us review the four research questions that we posed in Chapter 1 of this thesis. The following paragraphs describe how we address these research questions at the end of this thesis.

RQ1: How should behaviour- and performance properties of individual components be specified in order to enable automated composition of these properties into an analyzable model of a complete system?

We address the answer to this question by discussing two important aspects: the behavioural properties of components can be expressed by behaviour and process component models, and the performance properties can be specified by using a resource component model.

The introduced *behaviour and process component models* (see Sections 5.3.2 and 5.3.3) describe a component behaviour by specifying actions that the provided operations of the component are performing upon their invocation. These actions specify the sequence of "external" operations called by each specified operation. These "external" operations are provided by interfaces of neighboring (not known yet) components. At the time of specifying the component models, these neighboring components are not known (this information is only available at the component assembly time). However, a component developer does not need to have this information, because it is sufficient to know the signatures of the provided interfaces (of other components). These signatures are obtained from the required interfaces of the modeled component. As we mentioned in Chapter 2, the signatures of the provided and required interfaces of two bound components should match. The above-mentioned actions of an operation, specifying the sequence of calls of the "external" operations, can be considered as open connectors enabling the compositionality. At the time of creating a component assembly, when the neighboring components are known, the behaviour and performance properties of individual components represented in the models can be synthesized into system-wide properties, by use of these connectors.

The component *resource model*, described in Section 5.3.4 has been designed also for enabling compositionality. This model specifies hardware resource claims for each operation provided by a component. The processor claims are specified in terms of the *number of instruction cycles* that an operation requires from a processor to be executed. At the architectural phase, when it becomes known on which processor a component providing this operation is mapped, the number of instruction cycles can be recomputed into the *execution time* metric. This metric is then added to the system-wide model to annotate the processor usage of each operation involved in each task.

RQ2: How to combine the models of individual components into the model of a complete system in an automated way, such that the resulting system model can be analyzed with respect to the performance properties?

The global answer to this question is that we start from identifying the scenario and deployment models at the system level, in order to synthesize the involved component models in an automated way. The results lead to a list of synthesized tasks with the detailed behavioural and performance characteristics, which can be simulated to obtain the performance properties of

the system. The details of this approach are described in the following three paragraphs.

The architectural *scenario and deployment models* (see Section 5.4.2 and Section 5.4.3) are the input for the automated synthesis of the component models. The scenario model contains the specification of the composition of the service instances, the description of triggers, and the definition of parameter values for the parameter-dependent entities of component models. The deployment model defines the hardware architecture and the mapping of service instances onto the hardware nodes.

The proposed *synthesis algorithm*, described in Section 5.5 applies the models of the involved components to these scenario and deployment models. The algorithm identifies and generate tasks from the triggers (given in the scenario model) and the processes (specified in the process models of the involved components). Then, for each task, the algorithm recursively synthesizes the call graph of the involved operations, based on the call graphs of individual operations (these call graphs are provided by the behaviour/process models). Finally, for each element in the synthesized call graph, the algorithm computes the hardware resource-usage data (execution time or transfer size), based on data from the corresponding deployment, resource and hardware performance models.

This synthesis algorithm results in the constructed *executable system model* (see Section 5.4.4), which specifies a *task execution architecture*. This task execution architecture contains parameter-dependent data on the tasks running in the designed system, and the allocation of these tasks on the software and hardware components. The obtained tasks are suitable for different types of performance analysis (simulation, schedulability or hybrid), resulting in predictions of various performance properties of a system.

RQ3: How can architectural alternatives be compared and optimized with respect to multiple quality attributes?

In Chapter 6 we have presented a method for architecture optimization. The method proposes an iterative cycle containing the steps for (a) design of architectural alternatives, (b) predictions of multiple performance quality attributes of these alternatives, (c) Pareto-based trade-off analysis of the alternatives with respect to multiple performance attributes (e.g. task latencies, processor, bus and memory usage, performance sensitivity), and (d) identification of optimal alternatives and possible architecture transformations for further improvement of the optimal alternatives. This method is incorporated into the DeepCompass framework, supporting the above-mentioned four steps with the automating functionality of the CARAT toolkit.

RQ4: How can the assessment process of performance attributes be accelerated without a substantial reduction of the prediction accuracy?

We have proposed to accelerate the assessment process by (a) component models specified at relatively high level of abstraction and (b) usage of only critical scenarios for architecture analysis. Our models for a software component specify the behaviour or resource usage characteristics of the component at the level of operations (as an atomic entity), thereby omitting detailed programming constructs inside the operations such as variables, memory allocations, pointers, etc. This reduces modeling and analysis efforts, however, imposes certain inaccuracy in predicted results. In addition, the proposed usage of scenarios allows focusing only on those use cases and system execution paths which may hinder the performance requirements. These two proposals (high abstraction and critical scenarios) reduce the analysis time and effort. As indicated in Section 9.5, the average effort for a case study (including modeling and analysis of multiple alternatives for one iteration cycle) has proven to be only 36 hours.

Both the high-abstraction models of components and scenario-based analysis may potentially increase the prediction inaccuracy. Therefore, that was important to record the inaccuracy rate for each case study. The MPEG-4 case has shown the worst inaccuracy among the three studies. The deviations from the profiled measurements for the worst-case task latencies were within 30% and the deviations for the total processor utilization were within the 10% range. The highest prediction error has been recorded for the peak loads of the processor utilization. At some points of the timeline, the actual peaks have shown to be three times higher than the predicted peaks. This has happened due to the data-dependent processing nature of the MPEG-4 decoder. In the following iterations of the DeepCompass framework, we have addressed this issue by adding the facilities for parameter-dependent modeling to the component models.

10.3 Framework Limitations

Despite its benefits, the DeepCompass framework is not a "silver bullet" solution and has the following limitations.

The framework assumes that the *component models are already available* at the design phase. However, currently most of the existing CBSE components do not provide such models. Moreover, the specification of the required set of component models imposes some overhead for a component developer, despite the high abstraction level of these models.

The proposed analysis is performed at a *high abstraction level*. The provided set of modeling primitives allow specifying component implementation

only at high levels of abstraction (e.g. no modeling for pointers and asynchronous messages). The method does enable modeling the operation resource-claims that are characterized by probability distribution curves. Besides this, the method does not consider such important system-specific aspects as cache-misses, IO and memory access delays, as well the hard disc read-write latencies. We have imposed these limitations deliberately to reduce modeling and computational efforts. However, these limitations increase the performance prediction inaccuracy. To avoid this inaccuracy, the method can be extended with a 'cascaded' analysis, where another set of models specified at the medium or low abstraction level can be used for the post-analysis forming a final judgement of the performance properties.

The framework does not provide a *complete tool set for automated* multi-objective comparison of architectural alternatives and design space exploration. For the automation, the analytical activities required for the comparison and exploration stages should be formalized and implemented as a self-learning engine.

The scenario-based approach requires that *the architect has a good understanding* of the system-environment interaction aspects, and has some analytical skills in identifying the scenarios. We have addressed this issue by providing guidelines on scenario identification (see Section 8.3.3) obtained from the series of interviews with industrial architects.

10.4 Open Issues and Future Work

In order to improve the prediction accuracy, while keeping the modeling and computational efforts low, we aim to introduce a so-called *cascaded performance analysis* method. The method should provide a number of sub-methods at different abstraction levels. These sub-methods will provide a different set of modeling primitives and computation engines, but will be complementary and compatible with each other.

In addition to the high-abstraction sub-method (our scenario-based method), the medium- and low-abstraction sub-methods can be used as an extension, when the architectural overview is already defined and more accurate performance analysis is required. The low-abstraction sub-method should provide a set of highly-detailed models for software and hardware components, as well as semi-formal analysis engines. This provisioning should allow the architect to obtain the precise values for system performance properties. Such highly-detailed models may include a specification of the processor cache, IO and memory issues, as well as the description of intrinsic properties of a component implementation. As an obvious drawback, the low-abstraction sub-method would require substantial effort for modeling and computations.

In addition, we focus on the enhancement of the *automated architecture op-*

timization method, which we have outlined in Chapter 6. Firstly, we aim at the development of a self-learning engine, that would enable automated identification of architectural problem points in the available alternatives, based on the predicted values of their quality attributes. The found problem points should be converted by the engine into the rules/guidelines for further transformations of architectural models. Secondly, we need to formalize the specification of these rules, as well as the specification of possible transformations. Finally, we aim at the development of an evolution engine for design space exploration, that would realize the automated transformations of corresponding models, based on the defined transformation rules.

It is quite encouraging for the author that a considerable amount of the described modeling for components and assemblies – though with partly different terminology – has been recently adopted in the MPEG community (proceedings ICIP 2009, November). This has resulted in a working group that specifies components for Reconfigurable Video Coding (RVC). The purpose of these components is to quickly establish system simulations and performance analysis of a complete MPEG-4 AVC/H.264 coder that can be mapped on various platforms. It would be highly interesting to study and compare this approach with our method and come to joint improvements.

References

[1] G. Abowd, Analyzing Development Qualities at the Architecture Level: The Software Architecture Analysis Method, *Software Architecture in Practice*, Addison-Wesley, ISBN 0321154959, 1998.

[2] J. Andersson and D. Wallace. Pareto optimization using the struggle genetic crowding algorithm. *Engineering Optimization*, 34(6):623–643, Dec. 2002.

[3] F. Arbab. Abstract behavior types: a foundation model for components and their composition. *Sci. Comput. Program.*, 55(1-3):3–52, 2005.

[4] ARTiSAN Software Tools. *ARTiSAN Software Tools.* Public website, http://www.artisansw.com.

[5] P. Asterio, Integrating COTS software components into dependable software architectures, *Proc. of the 6th IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORCŠ03). Hokaido*, 2003.

[6] S. Becker, H. Koziolek, and R. Reussner, Model-based performance prediction with the palladio component model, *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, New York, NY, USA, ISBN 1-59593-297-6, 2007. ACM, pp. 54–65.

[7] A. Bertolino and R. Mirandola, Software performance engineering of component-based systems, *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, New York, NY, USA, ISBN 1-58113-673-0, 2004. ACM, pp. 238–242.

[8] E. Bini, G. C. Buttazzo, and G. M. Buttazzo. Rate monotonic analysis: The hyperbolic bound. *IEEE Trans. Comput.*, 52(7):933–942, 2003.

[9] P.E. Black. *Dictionary of Algorithms and Data Structures.* http://www.nist.gov/dads/HTML/preorderTraversal.html.

[10] Y. Bondarau. *CARAT-RTIE Performance toolkit, stable version.* http://www.win.tue.nl/trust4all/.

[11] E. Bondarev, M. R. V. Chaudron, H. Byelas, and P. H. N. de With, A toolkit for design and performance analysis of real-time component-based software systems, *ICSEA '06: Proceedings of the International Conference on Software Engineering Advances*, Washington, DC, USA, ISBN 0-7695-2703-5, 2006. IEEE Computer Society, pp. 4.

[12] E. Bondarev, M. R. V. Chaudron, and E. A. de Kock, Exploring performance trade-offs of a JPEG decoder using the deepcompass framework, *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, New York, NY, USA, ISBN 1-59593-297-6, 2007. ACM, pp. 153–163.

[13] E. Bondarev, M. R. V. Chaudron, and P. H. N. de With, Compositional performance analysis of component-based systems on heterogeneous multiprocessor platforms, *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, Washington, DC, USA, ISBN 0-7695-2594-6, 2006. IEEE Computer Society, pp. 81–91.

[14] E. Bondarev, M. R. V. Chaudron, and P. H. N. de With, CARAT: a toolkit for design and performance analysis of component-based embedded systems, *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, San Jose, CA, USA, ISBN 978-3-9810801-2-4, 2007. EDA Consortium, pp. 1024–1029.

[15] E. Bondarev, M. R. V. Chaudron, and P. H. N. de With. Multidimensional design space exploration for component-based architectures. *IEEE Transactions on Software Engineering*, pp. Accepted and under revision, Submitted in 2007.

[16] E. Bondarev, M. R. V. Chaudron, P. H. N. de With, and J. Muskens, Modelling of input-parameter dependency for performance predictions of component-based embedded systems, *EUROMICRO '05: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, Washington, DC, USA, ISBN 0-7695-2431-1, 2005. IEEE Computer Society, pp. 36–43.

[17] E. Bondarev, P. H. N. de With, and M. R. V. Chaudron, Towards predicting real-time properties of a component assembly, *EUROMICRO04: Proceedings of the 30th EUROMICRO Conference*, Washington, DC, USA, ISBN 0-7695-2199-1, 2004. IEEE Computer Society, pp. 601–610.

[18] E. Bondarev, P. H. N. de With, and M. R. V. Chaudron, A process for resolving performance trade-offs in component-based architectures, *Component-based software engineering, 9th International Symposium, CBSE6, Vasteras, Sweden*, Berlin, Germany, ISBN 3-540-35628-2, 2006. Springer, pp. 254–269.

[19] E. Bondarev, J. Muskens, P. H. N. de With, M. R. V. Chaudron, and J.J. Lukkien, Predicting real-time properties of component assemblies: A scenario-simulation approach, *EUROMICRO04: Proceedings of the 30th EUROMICRO Conference*, Washington, DC, USA, ISBN 0-7695-2199-1, 2004. IEEE Computer Society, pp. 40–47.

[20] E. Bondarev, M. Pastrnak, P. H. N. de With, and M. R. V. Chaudron, Predictable component-based software design of real-time MPEG-4 video applications, *Visual communications and image processing, VCIP05. Conference, Beijing , CHINE*. SPIE proceedings series, ISBN 0-8194-5976-3, 2005, pp. 254–269.

[21] G.C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Springer, October 2004.

[22] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: graphical editor and analyzer for timed and stochastic Petri nets. *Perform. Eval.*, 24(1-2):47–68, 1995.

[23] S. Chuprat and S. Salleh, A deadline-based algorithm for dynamic task scheduling with precedence constraints, *PDCN'07: Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference*, Anaheim, CA, USA, 2007. ACTA Press, pp. 158–163.

[24] J. E. Coffland and A. D. Pimentel, A software framework for efficient system-level performance evaluation of embedded systems, *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, New York, NY, USA, ISBN 1-58113-624-2, 2003. ACM, pp. 666–671.

[25] V. Cortellessa, A. Di Marco, and P. Inverardi, Integrating performance and reliability analysis in a non-functional MDA framework, *10th International Conference, FASE*, Vol. 4422 of *Lecture Notes in Computer Science*. Springer, ISBN 978-3-540-71288-6, 2007.

[26] R. Creps and P. Kogut. *Using DARPA Software Architecture Technologies to Design and Construct CORBA-based Systems.* Position Paper, http://www.objs.com/workshops/ws9801/papers/paper006.html.

[27] I. Crnkovic, M. Larsson, and O. Preiss, Vol. LNCS 3549, Concerning Predictability in Dependable Component-Based Systems: Classification

of Quality Attributes, in R. de Lemos (Ed.), *Architecting Dependable Systems III*, Springer-Verlag Berlin Heidelberg, ISBN 978-3-540-40539-9, 2005, pp. 257 Ű 278.

[28] DARPA. The Defense Advanced Research Projects Agency. *OWL-S: semantic markup for web services*. White Paper, 2003. Available from: http://www.daml.org/services/owl-s/1.0/owl-s.pdf.

[29] B. P. Douglass, *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[30] J. C. Laprie (ed), *Dependability: basic concepts and terminology*, Springer Verlag, 1992.

[31] D. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, Inc., New York, NY, USA, 2002.

[32] J. Fredriksson, K. Sandström, and M. Åkerholm, Optimizing resource usage in component-based real-time systems, *Proceedings of th 8th International Symposium on Component-Based Software Engineering (CBSE8)*, Vol. 3489 of *Lecture Notes in Computer Science(LNCS)*, May 2005, pp. 49–66.

[33] L.M. Garshol. *BNF and EBNF: What are they and how do they work?* http://www.garshol.priv.no/download/text/bnf.html.

[34] T. Genssler, Components for embedded software: the PECOS approach, *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, New York, NY, USA, ISBN 1-58113-575-0, 2002. ACM, pp. 19–26.

[35] T. Givargis and M. Palesi, Multi-objective design space exploration using genetic algorithms, *Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES-02)*, New York, 2002. ACM Press, pp. 67–72.

[36] S. S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE Trans. Dependable Secur. Comput.*, 4(1):32–40, 2007.

[37] K. Goseva-Popstojanova and K.S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Perform. Eval.*, 45(2-3):179–204, 2001.

[38] G.S. Graham. Guest editor's overview: Queuing network models of computer system performance. *ACM Comput. Surv.*, 10(3):219–224, 1978.

[39] V. Grassi, R. Mirandola, and A. Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *J. Syst. Softw.*, 80(4):528–558, 2007.

[40] Object Management Group. *UML Profile for Schedulability, Performance, and Time.* OMG Document ptc/2003-09-01, 2003, http://www.omg.org/cgi-bin/doc?ptc/2002-09-03.

[41] OMG Group. *Success Story. Boeing Commercial Airplanes Group.* http://www.omg.org/corba/industries/mfg/boeing.html.

[42] L. Grunske, Transformational patterns for the improvement of safety properties, *in Proc. of the Second Nordic Conference on Pattern Languages of Programs (VikingPLoP 03), Bergen Norge, 2003.*

[43] L. Grunske, Identifying "good" architectural design alternatives with multi-objective optimization strategies., in Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Ed.), *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006.* ACM, ISBN 1-59593-375-1, 2006, pp. 849–852.

[44] L. Grunske, B. Kaiser, and R. H. Reussner, Specification and evaluation of safety properties in a component-based software engineering process, Vol. 3778 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 2005.

[45] L. Grunske, P. A. Lindsay, E. Bondarev, Y. Papadopoulos, and D. Parker, An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems, *Architecting Dependable Systems IV, Lecture Notes in Computer Science*, Heidelberg, Germany, ISBN 978-3-540-74033-9, 2007. Springer Berlin, pp. 188–209.

[46] H. Hansson, SaveCCM - a component model for safety-critical real-time systems, *EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference*, Washington, DC, USA, ISBN 0-7695-2199-1, 2004. IEEE Computer Society, pp. 627–635.

[47] M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.

[48] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, System level performance analysis - the SymTA/S approach, *IEE Proceedings Computers and Digital Techniques*, 2005.

[49] S. Hissam. Predictable assembly of substation automation systems: An experiment report, 2002.

[50] S. Hissam, G. Moreno, J. Stafford, and K. Wallnau. Enabling predictable assembly. *J. Syst. Softw.*, 65(3):185–198, 2003.

[51] J. Horn, N. Nafpliotis, and D. E. Goldberg, A Niched Pareto Genetic Algorithm for Multiobjective Optimization, *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, Vol. 1, Piscataway, New Jersey, June 1994. IEEE Service Center, pp. 82–87.

[52] M. Hussein and M. Zulkernine. Intrusion detection aware component-based systems: A specification-based framework. *J. Syst. Softw.*, 80(5):700–710, 2007.

[53] A.T.H. Hutt, N. Donnelly, L.A. Macaulay, C.J.H. Fowler, and D. Twigger, Describing a product opportunity: A method for understanding the users' environment, in D. Diaper and R. R. Winder (Ed.), *People and Computers III*, British Computer Society, Cambridge: CUP, ISBN 0521351973, Feb 1988.

[54] IBM. *IBM Rational Rose Software.* Public website, http://www.ibm.com/rational.

[55] Intel. *Intel VTune Performance Analyzer.* White paper, http://www.intel.com/support/performancetools/vtune/sb/cs-009650.htm.

[56] ITEA Consortium. *Robocop: Robust Open Component Based Software Architecture.* Public Deliverables, http://www.hitech-projects.com/euprojects/robocop/deliverables.htm.

[57] J. Ivers and N. Sharygina. Overview of ComFoRT: A Model Checking Reasoning Framework, 2004.

[58] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley Professional, Feb 1999.

[59] R. Kazman, Tool support for architecture analysis and design, *2nd International Software Architecture Workshop (ISAW-2) at SIGSOFT '96*, New York, NY, USA, ISBN 0-89791-867-3, 1996. ACM, pp. 94–97.

[60] R. Kazman, L. Bass, M. Webb, and G. Abowd, SAAM: a method for analyzing the properties of software architectures, *ICSE '94: Proceedings of the 16th international conference on Software engineering*, Los Alamitos, CA, USA, ISBN 0-8186-5855-X, 1994. IEEE Computer Society Press, pp. 81–90.

[61] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. *Engineering of Complex Computer Systems, IEEE International Conference on*, 0:00–68, 1998.

[62] M. Kerholm, J. Carlson, J. Fredriksson, and H. Hansson. The SAVE approach to component-based development of vehicular systems. *J. Syst. Softw.*, 80(5):655–667, 2007.

[63] K. M. Khan and J. Han, Deriving systems level security properties of component based composite systems, *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, Washington, DC, USA, ISBN 0-7695-2257-2, 2005. IEEE Computer Society, pp. 334–343.

[64] S. Kim, J. Parka, and S. Hong, Vol. 48, Scenario-based multitasking for real-time object-oriented models, *Information and Software Technology*, Elsevier, ISBN 978-3-540-40539-9, Feb 2006, pp. 820–835.

[65] J. D. Knowles and D. W. Corne. Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy. *Evolutionary Computation*, 8(2):149–172, 2000.

[66] E. A. de Kock, YAPI: application modeling for signal processing systems, *DAC '00: Proceedings of the 37th conference on Design automation*, New York, NY, USA, ISBN 1-58113-187-9, 2000. ACM, pp. 402–405.

[67] E. A. de Kock, Multiprocessor mapping of process networks: a JPEG decoding case study, *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, New York, NY, USA, ISBN 1-58113-576-9, 2002. ACM, pp. 68–73.

[68] S. Kulturel-Konak, D. W. Coit, and F. Baheranwala. Reliability optimization of series-parallel systems using a genetic algorithm. *IIE Transactions*, 45(2):254–260, 2006.

[69] S. Künzli, L. Thiele, and E. Zitzler. Modular design space exploration framework for embedded systems. *IEE Proceedings - Computers and Digital Techniques*, 152(02):183–192, March 2005.

[70] C. Le and S. Hensley. Using COTS Components for Real-Time Processing of SAR Systems. 2004.

[71] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[72] Y. Liu, A. Fekete, and I. Gorton. Design-level performance prediction of component-based applications. *IEEE Trans. Softw. Eng.*, 31(11):928–941, 2005.

[73] D. Livolsi, T. O'Neill, J. Leaney, M. Denford, and K. Dunsire, Guided architecture-based design optimisation of CBSs, *ECBS 2006*. IEEE Computer Society, ISBN 0-7695-2546-6, 2006, pp. 247–258.

[74] C. A. Mattson and A. Messac. Pareto frontier based concept selection under uncertainty, with visualization. *Optimization and Engineering*, 6(1):85–115, 2005.

[75] T. Mowbray and R. Zahavi, , *Essential Corba*, John Wiley and Sons, New York, 1995.

[76] D. Nam and C. H. Park. Multiobjective Simulated Annealing: A Comparative Study to Evolutionary Algorithms. *International Journal of Fuzzy Systems*, 2(2):87–97, 2000.

[77] M. Nicholson, *Selecting a Topology for Safety-Critical Real-Time Control Systems*, PhD thesis, Department of Computer Science, University of York, 1998.

[78] D. M. Nicol, W. H. Sanders, and K. S. Trivedi. Model-based evaluation: From dependability to security. *IEEE Trans. Dependable Secur. Comput.*, 1(1):48–65, 2004.

[79] S. Nimmagadda, C. Liyanaarnchchi, A. Gopinath, D. Niehaus, and A. Kaushal, Performance Patterns: Automated Scenario-Based ORB Performance Evaluation, *Proceedings of the 5th Conference on Object-Oriented Technologies and Systems*, May 1999, pp. 15–28.

[80] G. Palermo, C. Silvano, and V. Zaccaria, A flexible framework for fast multi-objective design space exploration of embedded systems, *Proceedings of 13th International Workshop, PATMOS 2003, Torino, Italy, September 10-12, 2003*, Vol. 2799 of *Lecture Notes in Computer Science*. Springer, ISBN 3-540-20074-6, 2003, pp. 249–258.

[81] M. Pandey and D. Zappala, A Scenario-Based Performance Evaluation of Multicast Routing Protocols for Ad Hoc Networks, *Proceedings of the 6th Symposium on a World of Wireless Mobile and Multimedia Networks (WoWMoM)*, Los Alamitos, CA, USA, ISSN 0-7695-2342-0, May 2005. IEEE Computer Society, pp. 31–41.

[82] Y. Papadopoulos and C. Grante. Evolving car designs using model-based automated safety analysis and optimisation techniques. *Journal of Systems and Software*, 76(1):77–89, 2005.

[83] M. Pastrnak, P.H.N. de With, and J.L. van Meerbergen, QoS Concept for Scalable MPEG-4 Video Object Decodidng on Multimedia (NoC) Chips,

IEEE Trans. on Consumer Electronics, No. 4, Vol. 52, ISSN 0098-3063, Nov. 2006, pp. 1418–1426.

[84] D. Petriu, D. Amyot, and M. Woodside, Vol. 2708, Scenario-Based Performance Engineering with UCMNAV, *SDL 2003: System Design*, Springer Berlin / Heidelberg, Cambridge: CUP, ISBN 978-3-540-40539-9, Feb 2003, pp. 155.

[85] K. Philippe, The 4+1 View Model of architecture, *IEEE Softw.*, Vol. 12, Los Alamitos, CA, USA, ISSN 0740-7459, 1995. IEEE Computer Society Press, pp. 42–50.

[86] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Computers*, 55(2):99–112, 2006.

[87] F. Plasil, Enhancing component specification by behavior description: the SOFA experience, *WISICT '05: Proceedings of the 4th international symposium on Information and communication technologies*. Trinity College Dublin, ISBN 1-59593-169-4, 2005, pp. 185–190.

[88] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo. Reliability prediction for component-based software architectures. *J. Syst. Softw.*, 66(3):241–252, 2003.

[89] V. S. Sharma and K. S. Trivedi, Architecture based analysis of performance, reliability and security of software systems, *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, New York, NY, USA, ISBN 1-59593-087-6, 2005. ACM, pp. 217–227.

[90] D. Slama. *CORBA – The IT garage: Enterprise CORBA comes to Japan.* http://www.japaninc.com/cpj/features/features06a.html.

[91] SUN Microsystems. *PerfAnal: A Performance Analysis Tool.* White paper, http://java.sun.com/developer/technicalArticles/Programming.

[92] Arcticus Systems. *Rubus OS Reference Manual.* White Paper, http://www.arcticus.se.

[93] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley Professional, December 1997.

[94] G. Tesauro, W. E. Walsh, and J. O. Kephart, Utility-function-driven resource allocation in autonomic systems, *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, Washington, DC, USA, ISBN 0-7965-2276-9, 2005. IEEE Computer Society, pp. 342–343.

[95] L. Thiele, S. Chakraborty, and M. Naedele, Real-time calculus for scheduling hard real-time systems, *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, Geneva, Switzerland, 2000. pp. 101–104.

[96] Timesys. *LinuxLink by TimesysTM.* Public website, http://www.timesys.com/products/what.htm.

[97] M. Timmerman and L Perneel. *RTOS Evaluation Project.* Dedicated Systems Experts BN Document,DSE-RTOS-EVA-001b, 2005-09-13.

[98] S. Uchitel, G. Brunet, and M. Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Trans. Softw. Eng.*, 35(3):384–406, 2009.

[99] D. Urting, Y. Berbers, S. v. Baelen, T. Holvoet, Y. Vandewoude, and P. Rigole, A tool for component based design of embedded software, *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, Darlinghurst, Australia, Australia, ISBN 0-909925-88-7, 2002. Australian Computer Society, Inc., pp. 159–168.

[100] R. van Ommering, Vol. 33, , *The Koala component model for consumer electronics software*, IEEE Trans. Computer, Mar 2002, pp. 78–85.

[101] M. Verhoef. *The In-Car Radio Navigation case study.* White Paper, http://www.tik.ee.ethz.ch/ leiden05/data/pset/p2.pdf.

[102] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System architecture evaluation using modular performance analysis: a case study. *Int. J. Softw. Tools Technol. Transf.*, 8(6):649–667, 2006.

[103] X. Wu and M. Woodside. Performance modeling from software components. *SIGSOFT Softw. Eng. Notes*, 29(1):290–301, 2004.

[104] H. Ma X. Jin and Z. Gu, Real-time component composition using hierarchical timed automata, *QSIC '07: Proceedings of the Seventh International Conference on Quality Software*, Washington, DC, USA, ISBN 0-7695-3035-4, 2007. IEEE Computer Society, pp. 90–99.

[105] E. Zitzler, M. Laumanns, and L. Thiele, SPEA2: Improving the strength pareto evolutionary algorithm, in K. Giannakoglou, D. Tsahalis, J. Periaux, P. Papailou, and T. Fogarty (Ed.), *EUROGEN 2001. Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems*, Athens, Greece, 2002. pp. 95–100.

Curriculum Vitae

Yahor Bondarau received his MSc degree in Robotics and Informatics from the State Polytechnic University, Belarus Republic, in 1997. In 2003, he completed the post-master education program called OOTI of the Eindhoven University of Technology (TU/e), The Netherlands, for which he obtained a Professional Doctorate in Engineering (PDEng) in September 2003.

Currently, he is a Researcher at the Video Coding Architectures group, at the Electrical Engineering Faculty of the TU/e. He is focusing on the design of real-time component-based software, literally on the performance prediction of component-based systems on multiprocessor architectures. He was involved in several European research projects on software architectures (Space4U, Trust4All). He (co-)authored over 15 publications for international conferences, a book chapter on architecture optimization and he was a tutorial speaker for two IEEE conferences.