Metadata of the chapter that will be visualized in SpringerLink

Book Title	Intelligent Decision Making in Quality Management	
Series Title		
Chapter Title	Software Architecture Quality of Service Analysis Based on Optimization Models	
Copyright Year	2016	
Copyright HolderName	Springer International Publishing Switzerland	
Corresponding Author	Family Name	**Potena**
	Particle	
	Given Name	**Pasqualina**
	Prefix	
	Suffix	
	Division	Computer Science Department
	Organization	University of Alcalà
	Address	28871, Alcalà de Henares, Madrid, Spain
	Email	p.potena@uah.es
Author	Family Name	**Crnkovic**
	Particle	
	Given Name	**Ivica**
	Prefix	
	Suffix	
	Division	School of Innovation, Design and Engineering
	Organization	Mälardalen University
	Address	72123, Västerås, Sweden
	Email	
Author	Family Name	**Marinelli**
	Particle	
	Given Name	**Fabrizio**
	Prefix	
	Suffix	
	Division	Dipartimento di Ingegneria dell'Informazione
	Organization	Università Politecnica delle Marche
	Address	60131, Ancona, Italy
	Email	
Author	Family Name	**Cortellessa**
	Particle	
	Given Name	**Vittorio**
	Prefix	
	Suffix	
	Division	Dipartimento di Ingegneria e Scienze dell'Informazione, e Matematica
	Organization	Università dell'Aquila
	Address	67010, Coppito, AQ, Italy

	Email
Abstract	The ability to predict Quality of Service (QoS) of a software architecture supports a large set of decisions across multiple lifecycle phases that span from design through implementation-integration to adaptation phase. However, due to the different amount and type of information available, different prediction approaches can be introduced in each phase. A major issue in this direction is that QoS attribute cannot be analyzed separately, because they (sometime adversely) affect each other. Therefore, approaches aimed at the tradeoff analysis of different attributes have been recently introduced (e.g., reliability versus cost, security versus performance). In this chapter we focus on modeling and analysis of QoS tradeoffs of a software architecture based on optimization models. A particular emphasis will be given to two aspects of this problem: (i) the mathematical foundations of QoS tradeoffs and their dependencies on the static and dynamic aspects of a software architecture, and (ii) the automation of architectural decisions driven by optimization models for QoS tradeoffs.
Keywords (separated by '-')	Quality of service - Software architecture - Optimization - Medical informatics system

Chapter 14
Software Architecture Quality of Service Analysis Based on Optimization Models

Pasqualina Potena, Ivica Crnkovic, Fabrizio Marinelli and Vittorio Cortellessa

Abstract The ability to predict Quality of Service (QoS) of a software architecture supports a large set of decisions across multiple lifecycle phases that span from design through implementation-integration to adaptation phase. However, due to the different amount and type of information available, different prediction approaches can be introduced in each phase. A major issue in this direction is that QoS attribute cannot be analyzed separately, because they (sometime adversely) affect each other. Therefore, approaches aimed at the tradeoff analysis of different attributes have been recently introduced (e.g., reliability versus cost, security versus performance). In this chapter we focus on modeling and analysis of QoS tradeoffs of a software architecture based on optimization models. A particular emphasis will be given to two aspects of this problem: (i) the mathematical foundations of QoS tradeoffs and their dependencies on the static and dynamic aspects of a software architecture, and (ii) the automation of architectural decisions driven by optimization models for QoS tradeoffs.

Keywords Quality of service · Software architecture · Optimization · Medical informatics system

---

P. Potena (✉)
Computer Science Department, University of Alcalà, 28871 Alcalà de Henares, Madrid, Spain
e-mail: p.potena@uah.es

I. Crnkovic
School of Innovation, Design and Engineering, Mälardalen University, 72123 Västerås, Sweden

F. Marinelli
Dipartimento di Ingegneria dell'Informazione, Università Politecnica delle Marche, 60131 Ancona, Italy

V. Cortellessa
Dipartimento di Ingegneria e Scienze dell'Informazione, e Matematica, Università dell'Aquila, 67010 Coppito, AQ, Italy

C. Kahraman and S. Yanik (eds.), *Intelligent Decision Making in Quality Management*, Intelligent Systems Reference Library 97,
DOI 10.1007/978-3-319-24499-0_14

Layout: T1 Standard Unicode Book ID: 312194_1_En Book ISBN: 978-3-319-24497-6
Chapter No.: 14 Date: 16-9-2015 Time: 6:22 pm Page: 2/26

14.1 Introduction

The presence in the market of standard off-the-shelf components/services has drastically changed in the last decade the development process of component-based and service-based systems (as claimed, e.g., in Szyperski 2002).

A software system today is no more conceived as a product to be built "from scratch"; rather software engineers aim at building a system where several software units—components/services, each satisfying a certain number of requirements—interact each other and with users to accomplish the tasks required.

Requirements can be partitioned in functional and non-functional. The former concerns "what" the software has to do, while the latter concern "how" the software works. In a service-oriented architecture, in practice, functional requirements determine the services the system should provide, whereas non-functional requirements (that determine the Quality of Service of the system), are constraints on the services offered by the system, such as timing constraints or constraints on the development process (Sommerville 2004).

The properties (functional and non-functional) of the final software product therefore heavily depend on (i) the properties of the reused software units and those of newly built software units, as well as on (ii) the way these software units are assembled (i.e. the software architecture).

In the last years several research efforts have been devoted to the definition of models representing dependencies between non-functional properties of single elements and the properties of the whole system.

External properties (i.e., system attributes) are functions of both internal properties (i.e. attributes of elementary components or services) and other factors, such as system architecture or usage profile. Developers must therefore address how the integrated system inherits attributes of elementary parts. For example, if you integrate several high performance or high-reliability components, what can you say about the performance or reliability of the system as a whole? Similarly, if you integrate a combination of low and high-quality components, how can you assess and improve the resulting system's quality? (Brereton and Budgen 2000). The formulation of such models is not easy due to complex relationships between components that may be hard to express in a closed form.

Component-Based Software Engineering (CBSE) and Service-Oriented Software Engineering (SOSE) are the most dominant disciplines that deal with problems of building software systems based on (reused and newly built) components/services (Breivold and Larsson 2007). The ability to predict QoS of a software architecture has to be supported by a large set of decisions arising from several phases of the software lifecycle that span from design, through implementation-integration, to adaptation phase. However, due to the different amount and type of information available, different prediction approaches should be introduced in each phase.

In the design and implementation phase, elementary software units are typically selected and verified/tested alone or in combination with other (selected) software units at the aim of choosing the combination that best fits the goals. In fact, it is well known (Wallnau and Stafford 2002) that even if isolated components correctly work, an assembly of them may fail due to not immediately apparent dependencies and relationships, such as shared data and resources. Besides, since the software units always have to be deployed on an hardware platform, the best mapping of software onto hardware with respect to certain criteria (e.g. performance of the whole system) has to be considered as well. Finally, an existing software unit (or a set of units) would be replaced and/or new units would be adopted in the maintenance phase, e.g., when the requirements of the system evolve or when the vendor of the component releases an updated version,[1] while keeping other units unchanged.

On the basis of the above considerations, it is evident that the architectural decisions must be carefully carried on taking into account non-functional properties (besides functional ones). In fact, functionally equivalent software units (to be used for replacing existing software units or to be added to the system) may heavily differ in their non-functional properties, affecting in this way the QoS at various extents. We hence forth refer to functionally equivalent software units that differ for their non-functional properties as to *instances*.

One of the most prominent characteristic of a software unit is its cost. In general, the cost of an in-house developed component depends (among others) on the development and testing effort required to deliver the component. On the other hand, the cost of a purchased component depends (among others) on its buying price and on the effort needed to adapt it to the working context.

The non-functional properties and the cost of a software unit are typically tied. Indeed, components and services with high quality value in general result to be the more expensive ones. Hence there is an intrinsic trade-off between the cost of a software product, which results from the costs of its elementary elements plus, e.g., the cost for component/service adaptation, and its quality that result from both the non-functional properties of its elementary elements and other characteristics such as the architecture of the system.

In general, the definition of architectural decision criteria based on non-functional properties is not easy. In fact, an elementary unit could be the best one with respect to a certain property, but at the same time it could be either too expensive or not compliant with possibly constraints on other non-functional properties.

Due to the complexity of addressing non-functional criteria in the architectural decision-making process, and given the extremely high number of parameters to consider in order to achieve a (near-) optimal decision, the introduction of quantitative methods and automatic tools would help the software engineers to raise their focus from a human-based search to a machine-based search.

---

[1] A deeper discussion on the peculiarities of the component selection activity within each phase of a development process can be found in Cortellessa et al. (2008).

Quantitative methods find their natural definition in the field of optimization. An optimization model allows, for example, to find a solution that minimizes the cost of a software system while satisfying requirements that can be expressed as a set of mathematical constraints. Optimization techniques have been already proposed and used for the analysis of QoS tradeoffs of a software architecture (Aleti et al. 2013). In Sect. 14.2 we discuss this aspect.

In this chapter we focus on the optimization-based modeling and analysis of software architecture QoS tradeoffs. A particular emphasis will be given to two aspects of these tasks: (i) the mathematical foundations of QoS tradeoffs and their dependencies on the static and dynamic aspects of a software architecture, and (ii) the automation of the architectural decision-making process driven by QoS tradeoff optimization models.

In particular, we present a general optimization model that minimizes the total costs subject to constraints on the level quality of the software architecture. The model can be adopted in (specialized for) one of the lifecycle phases by leveraging available information and parameters, the level of detail of which obviously increases as the development progresses. Then, each specialized form of the general model can be either separately used and solved, if required in a certain lifecycle phase, or used in pipeline feeding with each other, as we will show in our example.

In the context of a waterfall development process, we implement three models: one for the architectural design (i.e. the software architecture driven model applicable before the release of a system), one for the implementation/deployment phase (we show how the QoS of a software architecture depends on the hardware architecture), and one for the maintenance phase (i.e. the software architecture driven model applicable after the release of a system).

In order to show the usefulness of our approach, we run these models on an example coming from the domain of medical information systems. We also study the sensitivity of the solutions to changes of parameters; we analyze, in particular, the behavior of the system costs at varying of non-functional requirements, see Potena et al. (xxx).

Although here we describe the phases and interactions that fit well in a waterfall approach and show how our models can be employed in such a context, our approach is not limited to the waterfall design process only. Different paradigms can be considered, provided that the interactions between phases are properly taken into account. Indeed, the interactions between phases may change among different design approaches. For example, the interaction between the requirements and design phases will repeat when performed within agile, iterative or incremental development frameworks. In such cases the decision-making process would converge faster, e.g., due the know-how acquired and/or the activities performed in the previous iterations of the process. Also, in case of selecting new software units for new requirements, potential compatibility problems with existing units can be already recognized in the early phase of the process.

Our major contribution is to show how effectively optimization modeling techniques can capture relevant aspects of the architectural decision-making process in different lifecycle phases, thus representing a very relevant support for the software engineer's tasks.

All the proposed models belong to the class of mixed-integer nonlinear programming problems and therefore can be solved by means of exact and heuristic optimization techniques such as spatial branch-and-bound (Belotti et al. 2009) and Tabu-search. Although such problems are generally hard to solve due to non-linearities and integrality, they can be handled by common solvers (we used LINGO http://www.lindo.com in our computational assessment) since usually they are small for most of the common software domains. For large scale problems, however, search-based techniques, e.g., Tabu-search or genetic algorithms (Blum and Roli 2003), can be successfully adopted. Indeed, such techniques have been applied for obtaining solutions for several problems in the software engineering domain, from requirements and project planning to maintenance and re-engineering (Harman et al. 2012).

The chapter is organized as follows. In Sect. 14.2 we present related works and discuss the novelty of our contribution. In Sect. 14.3 the most common problems encountered for QoS tradeoffs analysis are discussed, and in Sect. 14.4 we introduce the general formulation of an optimization model for such kind of analysis. Section 14.5 describes the distributed medical informatics system adopted as example. Sections 14.6, 14.7 and 14.8 detail the optimization models and their application to the example for the architectural design, maintenance, and implementation/deployment phases, respectively. Finally, conclusions are delineated in Sect. 14.9. In Potena et al. (xxx) we have collected all the further details that are not strictly necessary for this chapter understanding.

14.2 Related Work

A quite extensive collection of papers on decision-making processes across lifecycle phases and on methods/tools able to predict and evaluate the QoS of a software architecture can be found in literature. Decision-making frameworks have been introduced to facilitate the reasoning process for different goals and from different perspectives. For example, software architecture has been used for documenting and communicating design decisions and architectural solutions (Clements et al. 2011). However, being the focus of this chapter on the QoS tradeoffs' analysis of a software architecture, we report only papers that present similar criteria for this task. This helps us to clearly describe, at the end of this section, the novelty of this chapter with respect to the existing related work.

Several qualitative methods have been proposed in order to explicitly analyze the impact of architectural decisions on system quality, among which the well-known Architecture Tradeoff Analysis Method (Kazman et al. 1998) and Cost Benefit Analysis Method (CBAM) (see, for example, the survey in Breivold et al. 2012). These evaluation techniques suffer of some weaknesses that mainly are the subjective point of view of the analysts and the heavyweight process, which requires many steps and intense participation of stakeholders (Kim et al. 2007).

In order to overcome these limitations, qualitative attributes are transformed into quantitative figures, e.g., see the Multi-Criteria Decision Analysis (MCDA) technique that combines Analytic Hierarchy Process (AHP) and CBAM (Lee et al. 2009; Kim et al. 2007). Other common techniques such as AHP and Weighted Scoring Method (WSM) are used, for example, by component selection approaches (Kontio 1996). In particular, WSM estimates how to modify a software architecture, e.g. by introducing a different COTS component, with respect to a set of weighted criteria. The score of the change is calculated by the weighted sum of the criteria values. Alternatively, AHP suggests to define a hierarchy of criteria. Modification choices are compared in pairs and finally ranked on the basis of a score that combines the results of the comparison. Both the above methods come with serious drawbacks: the combinatorial explosion of the number of pair-wise comparisons, the need of extensive a priori preference information, and the highly problematic assumption of linear utility functions. Optimization techniques may solve some of these drawbacks because, in general, they do not need any weighting and/or ranking of the evaluation criteria (Neubauer and Stummer 2007).

Several research efforts have also been devoted in the last years to the designing of optimization methods for the analysis of software architectures (a quite extensive list of these approaches can be found in Aleti et al. 2013). Mostly depending on the lifecycle phase, different types of decisions and quality analysis methods are considered. Typically the decisions span from the service/component selection (e.g., Cardellini et al. 2012; Yang et al. 2009) through the deployment of components/services (e.g., Malek et al. 2012; Vinek et al. 2011) to the application of recurring software designs solutions[2] (e.g., Mirandola and Potena 2011). All these approaches basically provide guidelines to automate the search for an optimal architecture design based on the QoS tradeoffs.

The QoS tradeoffs analysis of such approaches basically is based on simple optimization models (see, e.g., Cortellessa et al. 2010) or multi-objective optimization models that, for example, maximize both reliability and performance (see, e.g., Cardellini et al. 2012). Different techniques are used to solve such optimization models, such as metaheuristic techniques, integer programming, or a combination of both (see, for example, surveys Harman et al. 2012; Aleti et al. 2013). For example, the work in Grunske (2006) shows how evolutionary algorithms and multi-objective optimization strategies, based on architecture refactorings, can be implemented to identify architecture designs, which can be used as an input for architecture tradeoffs analysis techniques.

Usually the goal of the existing approaches is to predict and/or analyze QoS attribute, like performance or reliability, starting from the architectural description of the system, or to select the architecture of the system, among a finite set of

---

[2] They provide a generic solution to address issues pertaining to quality attributes, like the architectural tactics (Vinek et al. 2011).

candidates, that better fulfill the required quality. In our previous works (Cortellessa et al. 2010; Potena 2013), we have addressed the problem of system quality from a different point of view: starting from the description of the system and from a set of new requirements, we devise the set of actions to be accomplished to obtain a new architecture. This is able to fulfill the new requirements with the minimum cost based QoS tradeoffs (i.e., reliability vs. availability, and vs. performance).

Other challenges related to the quality analysis are represented by the lots of different type of uncertainties that can be faced during the decision-making process. The specification of the effect of architectural decisions on goals (e.g., functional or nonfunctional requirements) is a difficult task. As a consequence, the process of making early architectural choices is a risky proposition mired with uncertainty (Esfahani et al. 2012). Several interesting approaches have been introduced in order to make the uncertainty explicit and using it to drive the production process itself (see, for example, Esfahani et al. 2012; Autili et al. 2011; Ghezzi et al. 2013) some of which are detailed below). In particular, for the design time (early phases of the software development process), the GuideArch framework (Esfahani et al. 2012) guides the exploration of alternative architectures under uncertainty by exploiting fuzzy mathematical methods. GuideArch allows to compare alternative architectures with respect to system's properties (like cost and battery usage). The ADAM (Adaptive Model-driven execution) framework (Ghezzi et al. 2013), based on probability theory and probabilistic model checking, supports the development and execution of software that tolerates manifestations of uncertainty by self-adapting to changes in the environment, trying to do its best to satisfy certain non-functional requirements (i.e., response time and the faulty behavior of components integrated in a composite application).

Research efforts have also been spent in order to deal with parameters' uncertainty (Meedeniya et al. 2012; Doran et al. 2011; Wang et al. 2012; Wiesemann et al. 2008). In particular, in Meedeniya et al. (2012), a robust optimization approach allows to deal with the impact of inaccurate design-time estimates of parameters. A Bayesian approach has been introduced in Doran et al. (2011), in order to systematically consider parametric uncertainties in architecture-based analysis. In Wang et al. (2012), the propagation of a single parameter's uncertainty on the overall system reliability estimation is analyzed. Finally, in Wiesemann (2008), the stochastic programming is exploited to support the service composition under quality attributes tradeoffs. In particular, the service composition problem is formulated as a multi-objective stochastic program which simultaneously optimizes some quality-of-service parameters (i.e., workflow duration, service invocation costs, availability, and reliability).

The originality of this chapter mainly consists in showing how effectively optimization modeling techniques can capture relevant aspects of the architectural decision making process in different lifecycle phases, thus representing a very relevant support for the software engineers' decisions. Our overall approach of embedding optimization models for different lifecycle phases is, at the best of our knowledge, the first example of an integrated framework for supporting developers' decisions based on cost/QoS tradeoffs during the whole software development process. Moreover, our optimization models are not tied to any particular development process as well as they do not depend on the specific application domain.

14.3 Typical Problems of QoS Tradeoffs Modeling

There are some limitations in the analytical formulation of non-functional aspects of components/services-based software systems mostly due to the intrinsic complexity of the component/service inter-relationships. Here below we summarize the major points.

In general, the quality attributes (such as response time and availability) depend on many observable parameters (such as size of messages exchanged, number of function points, etc.) that might be tightly correlated with each other. Some assumptions are typically made in order to keep as simple as possible the model formulation. For example, most reliability models for systems composed by basic elements (e.g. objects, components or services Goseva-Popstojanova and Trivedi 2001; Krka et al. 2009; Immonen and Niemelä 2008; Becha and Amyot 2012) assume that the elements are independent, namely the models do not take into account the dependencies that may exist between elements. They assume that the failure of a certain element provokes the failure of the whole system. What is basically neglected under this assumption is the error propagation probability, which in several real domains (such as control systems) is not an issue, because component/service errors are straightforwardly exposed as system failures. In order to relax such an assumption, an error propagation model must be introduced (see, for example, the reliability model for service-based systems introduced in our previous work Cortellessa and Potena 2007).

Also the non-functional properties are tightly correlated, and often depend on each other. In fact, some conflicts could exist among quality attributes (Boehm and In 1996), e.g., suitable tradeoffs between modifiability and performance have to be provided while building a software architecture, as remarked in Lundberg et al. (1999).

Sometimes the providers of pre-existing components/services are not able to come up with the exact values of some non-functional properties, and simply get a set of ranges over which the values may lie. For example, the component reliability for a given component cost is usually specified over a range based on prior experience (Gokhale 2007). If only ranges are available, then optimization can be performed on a parametric model, i.e., a model with some parameters ranging within provided limits, in order to observe the trend and sensibility of solutions.

In other cases, the information provided by vendors are not enough to estimate the non-functional properties of a given component/service since some of its parameters (e.g. cost or reliability) may be characterized by a not negligible uncertainty. In the case of component reliability, the propagation of such uncertainty is analyzed by Goseva-Popstojanova and Kamavaram (2004), Dai et al.

(2007). However, it was out of the scope of this chapter to deal with this kind of sensitivity analysis.

The reliability estimation methods typically deal with the operational profile (Musa 1993; Chandran et al. 2010) which is another factor that brings uncertainty in QoS analysis. In fact, the operational profile of the system is in general different from the one adopted to estimate the non-functional properties of elementary components/services. As remarked in Becker and Koziolek (2005), no standard model are available for describing the operational profile and hence it is necessary to take into account the transformations that the components may provide on it. "Inputs on the provided interfaces of a component are transformed along the control flow down to the required interfaces. Thus, the provided interfaces of subsequent components connected with the required interfaces receive a different operational profile than the first component. The transformations form a chain through the complete architecture of components until the required interfaces of components only execute functions of the operating system or middleware" (Becker and Koziolek 2005). However, if the operational profile of the system is not (fully) available at the design phase, the domain knowledge and the information provided by the software architecture in general are sufficient for estimating it, as suggested in Roshandel and Medvidovic (2007) or in Musa (1993).

The integration of components/services often entails mismatches whose handling cost should be included into the QoS tradeoffs modeling. Several approaches have been introduced to deal with the mismatches problems (e.g., see Park 2006; Younas et al. 2005 for the integration of web services in distributed system). For solving a mismatch between a requirement and a pre-existing software unit, different actions are possible, and different existing works could be exploited, such as the approach presented in Mohamed et al. (2007), which supports the resolution of mismatches during and after a COTS selection process by using an optimization model.

As far as concerns the non-functional requirements, the task of handling mismatches between the properties of single components/services and the quality required for the whole system is even harder than one for the functional mismatches, e.g., sometimes the improvement of a single software unit could not affect the quality of the whole system. Clearly, closed formulas for estimating the quality of the system as a function of the properties of components/services would be very helpful, but many problems have to be faced for defining them.

14.4 A General Formulation for Architectural Decisions Versus Quality

In this section we propose a general optimization model that helps developers to make the QoS tradeoffs analysis of a software architecture.

Let $S = \{u_1, \ldots, u_n\}$ be a software architecture made of n software units $u_i (1 \leq i \leq n)$ the composition of which results in services that the system offers to users.

Since the proposed model may support different lifecycle phases, we adopt a general definition of software unit: it is a self-contained deployable software module containing data and operations, which provides/requires services to/from other elementary elements. A unit instance is a specific implementation of a unit.[3] For each unit u_i, let J_i be the set of instances available by vendors and $\bar{J}_\imath$ the set of possible options for developing the instance in-house. Let u_{ij} be the jth instance of $J_i \cup \bar{J}_\imath$.

The analysis of the QoS tradeoffs is a broad decision-making process that consists of a set of actions aiming to modify the static and dynamic structure of the software architecture. The decisions within the different life-cycle phases are basically related to the following software actions:

1. *Introducing new software units*: One or more new software units may be embedded into the system.[4] We call *NewS* the set of new available software units that can provide different functionalities.
2. *Replacing existing unit instances with functionally equivalent ones available on the market*: The employed instance u_{ik} of a software unit u_i may be replaced with an element of the set J_i, i.e., with of the instances available for it on the market (e.g. a Commercial-Off-The-Shelf (COTS) component/web service). We assume that all the instances in J_i are functionally compliant with u_{ik}, i.e., each of them provides at least all services provided by u_{ik} and requires at most all services required by u_{ik}.[5] The instances in J_i may differ from u_{ik} for cost and quality attribute (e.g. reliability and response time).
3. *Replacing existing unit instances with functionally equivalent ones developed in-house*: An existing instance of a software unit u_i may be replaced with one developed in-house. Developers could opt for different building strategies resulting in different in-house instances, i.e., the elements of the set $\bar{J}_\imath$. The values of quality attributes of such optional instances (e.g., reliability, response time) could vary due to the values of the development process parameters (e.g. experience and skills of the developing team).
4. *Modifying the interactions among software units in a certain functionality*: The system dynamics may be modified by introducing/removing interactions among software units within a certain functionality.

---

[3] The optimization model can work for any semantics given to software units under the condition that the parameters are associated to the correct units. The only difference, of course, is in the techniques needed to estimate the model parameters, but this is out of the scope of this chapter.

[4] Notice that such type of action has to be associated to another action that indicates how this unit interacts with existing units, therefore it modifies the interactions within certain functionalities (see last type of software action).

[5] As remarked in Cortellessa et al. (2010), such an assumption could be relaxed by introducing integration/adaptation costs.

Clearly, the system quality heavily depends on the hardware features, e.g., response time decreases as the processing capacity improves, and therefore decisions on software architecture must also take into account the decisions on the hardware characteristics of the system. Hardware decisions typically span from the deployment of software units on hardware nodes through to modify the characteristics of the underlying hardware resources (e.g., CPU, disk, memory, network throughput, etc.) to introducing/removing connection links among hardware nodes.[6] Indeed, depending on the adopted engineering paradigm (e.g., CBSE or SOSE), different types of hardware changes may be performed. For example, as explained in Mirandola and Potena (2011), in the SOA domain, due to the fact that the services are not acquired in terms of their binaries and/or source code, but they are simply used while they run within their own execution environment (that is not necessarily under the control of the system using them), hardware changes can be suggested by the service providers.

Optimization Model Formulation

All the above actions can be modeled by decision variables that describe the software architecture instances selection process. In particular, let $x_{ij}(1 \leq i \leq n, j \in J_i \cup \bar{J}_\iota)$ be the binary variable that is equal to 1 if the instance j is chosen for thesoftware unit i, and 0 otherwise. Moreover, let $z_h(1 \leq h \leq |NewS|)$ be the binaryvariable that is equal to 1 if the new software units h is chosen and 0 otherwise.

Let us suppose to analyze the system on the base of p quality attributes (such as cost, response time, availability, etc.). Suppose moreover that each attribute of any softwareunit depends on the value of parameters α_i^k's, β_i^k's, and γ_{ij}^k's, where (i) the vector α_i^k describes the (at most) u software architecture observable parameters, e.g., the average number of invocations of a software unit within the execution scenarios considered forthe software architecture, (ii) the vector β_i^k contains the (at most) v hardware observableparameters, e.g., the processing capacity of the node hosting the software unit, thatis measured, for example, as the average number of instructions per second that there source can execute, and (iii) the vector γ_{ij}^k represents the (at most) w features of theimplementation of u_i, e.g., the reliability of the instance used for replacing the existingunit. For the k quality attributes of a provided instance, the value of the features γ_{ij}^k's is assumed to be either given from the software unit provider or estimated from the customer. On the contrary, for an in-house developed instance the γ_{ij}^k's can be predicted by considering variables of the decision planning. For example, in Sect. 14.6, we express the reliability of an in-house instance as a function of a variable representing the amount of testing N_i^{tot} to be performed on that instance.

Let $\Gamma_k : \mathbb{R}^u \times \mathbb{R}^v \times \mathbb{R}^w \to \mathbb{R}(\bar{\Gamma}_k : \mathbb{R}^u \times \mathbb{R}^v \times \mathbb{R}^w \to \mathbb{R})$ be the function that, on the base of the above parameters, returns the value of the kth quality attribute

---

[6] A deeper discussion on the hardware changes can be found in Mirandola (2011).

$(1 \leq k \leq p)$ of an existing (new) software unit. In particular, let $\Lambda_{ij}^k = \Gamma_k\left(\alpha_i^k, \beta_i^k, \gamma_{ij}^k\right)$ the value of the *k*th attribute of the provided/in-house instance u_{ij}.

For sake of readability, we introduce here a formulation without correlations among Γ_k's, where each quality attribute does not affect other attributes and a self-contained analytical expression can be formulated for it. Obviously this is not always true, as it depends on the considered quality attributes and the model complexity. If quality attributes have to be correlated (Bass et al. 2002) (e.g., when perform ability is considered) then additional constraints may be needed, which can be expressed as *contingent decisions* (Jung and Choi 1999).

We can represent the value of the *k*th quality attribute of the *i*th existing softwareunit as a function of the decisional strategy **x**:

$$\theta_i^k = \sum_{j \in \bar{J}_i \cup J_i} \Lambda_{ij}^k x_{ij} \qquad (14.1)$$

Similarly, we can represent the value of the *k*th quality attribute of the *h*th newsoftware unit as a function of the decisional strategy **z**:

$$\theta_h^{-k} = z_h \bar{\Gamma}_k\left(\alpha_i^k, \beta_i^k, \gamma_{ij}^k\right) \qquad (14.2)$$

Let $G_k : \mathbb{R}^n \times \mathbb{R}^{|News|} \rightarrow \mathbb{R}$, with $(1 \leq k \leq p)$, be the function that returnsthe *k*th quality attribute of the whole system on the base of the same attributes of each existing/new software unit. And let us assume (without loss of generality) that the values of each quality attribute *k* are constrained to be above a lower threshold value Θ^k. Assume, moreover, that the cost is the first quality attribute, i.e., $\theta_i^0 (\bar{\theta}_i^0$ express the cost of the existing (new) software units. Finally, let $Cost : \mathbb{R}^n \times \mathbb{R}^{|NewS|} \rightarrow \mathbb{R}$ be the cost function of the whole system that clearly depends on the costs of all the existing (new) software units. Different cost models could be used to define *Cost*, e.g., it may also include the potential costs of software unit adaption (i.e. the glue ware). For the sake of readability, we introduce here a formulation without correlation between the software unit costs and the other software/hardware quality attributes.

The general formulation of the optimization model for the QoS tradeoffs analysis is given by:

$$\min_{x,z} Cost(\theta^0, \bar{\theta}^0) \qquad (14.3)$$

s.t.

$$G_k\left(\theta^0, \bar{\theta}^0\right) \geq \Theta^k \quad \forall k = 1..p$$

$$\sum_{j \in \bar{J}_i \cup J_i} \Lambda_{ij}^k x_{ij} = \theta_i^k \quad \forall k = 1\ldots p, \quad \forall i = 1\ldots n$$

$$z_h \bar{\Gamma}_k\left(\alpha_h^k, \beta_h^k, \gamma_h^k\right) = \bar{\theta}_i^k \quad \forall k = 1\ldots p, \quad \forall h = 1\ldots|NewS|$$

$$\sum_{j \in \bar{J}_i \cup J_i} x_{ij} = 1 \qquad \forall i = 1\ldots n$$

$$x_{ij} \in \{0, 1\} \quad \forall i = 1\ldots n, \quad \forall j = 1\ldots p$$

$$z_h \in \{0, 1\} \quad \forall h = 1\ldots|NewS|$$

Other constraints (e.g., equations to predict α_i^k's and β_i^k's).

14.5 An Example: A Distributed Medical Informatics System

In this section we describe the main features of an example that we will use for illustrating the application of our approach (see Sects. 14.6, 14.7 and 14.8). For sake of readability, a description of the high-level structure of the system, together with all the details on the models, i.e., the meaning of additional parameters and constraints and on the computational results is available in Potena et al. (xxx).

We have considered the distributed medical informatics system described by Yacoub et al. (1999) mainly because its features allow us to show how effectively optimization modeling techniques can capture relevant aspects of the architectural decision making process in different lifecycle phases. Shortly, medical institutions need in general to exchange information, e.g., medical images, between each other. Actually, they form a client/server system where the *AE Client* subsystem is connected to the *AE Server* subsystem by the *Network* subsystem. The communication between the entities of the system is performed using Digital Imaging and Communication in Medicine (DICOM) standard,[7] which is typically used for producing, processing and exchanging medical images: "The DICOM specifies the transport and presentation layer for a network protocol as DICOM Upper Layer (*DICOM UL Client* and *Server* subsystems)" (Yacoub et al. 1999).

In the following sections, we will analyze the three scenarios identified by Yacoub et al.: We will consider *AE Client*, *Network*, *AE Server*, *DICOM UL Client* and *Server* subsystems as architectural elementary elements of the system. Moreover, we will suppose that *Network* subsystem does not identify all the network, but a component which is deployed along the network.

---

[7] http://medical.nema.org/.

14.6 Architectural Design Phase

14.6.1 Before Release (Platform Independent)

For the design phase, the general optimization model (3) is instantiated with a mathematical formulation that stems from our previous work in the context of component based software (Cortellessa et al. 2006). Specifically, we consider the following architectural decisions: (i) replacing existing unit instances with functionally equivalent ones available on the market, and (ii) replacing existing unit instances with functionally equivalent ones developed in-house.

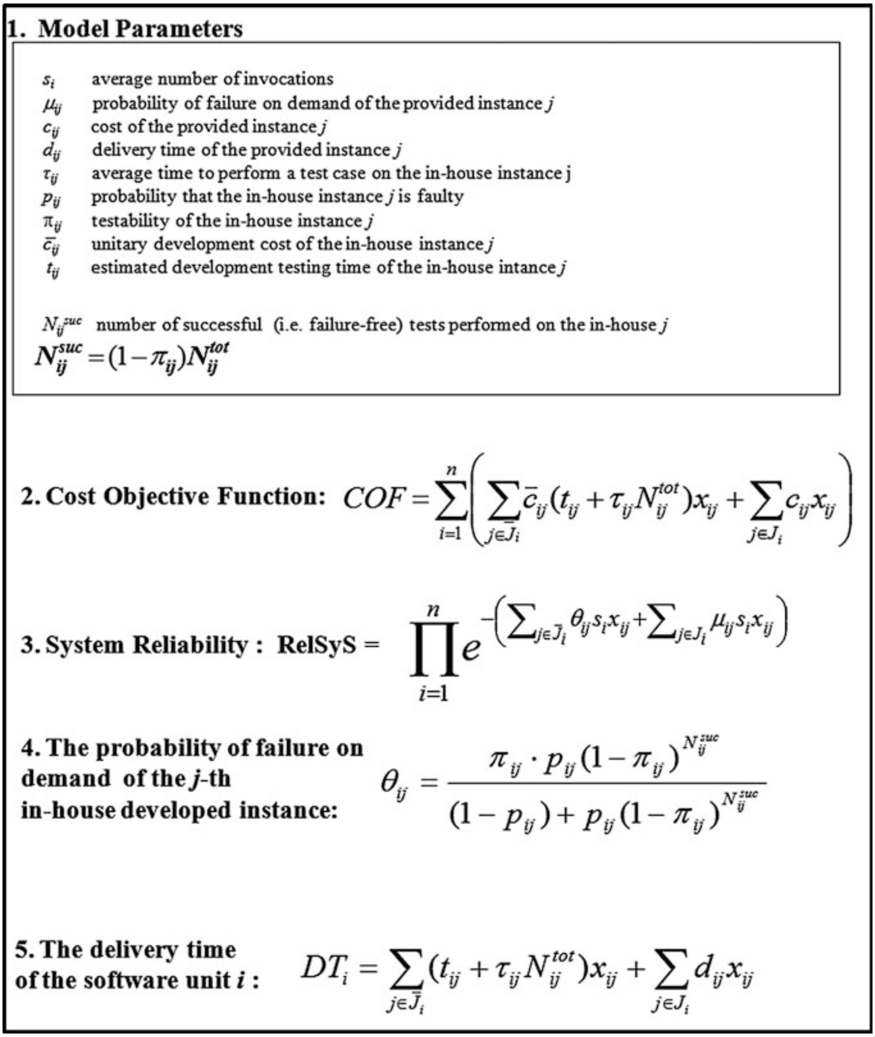
We report the model formulation by plugging the problem in a general application domain, where the build-or-buy decisions refer to general software unit rather than components. Additional constraints on delivery time and reliability of the system are considered, and decision planning variables associated to the amount of testing to be performed on each in-house instance are introduced.

Our model definition makes the following significant assumptions. (i) We assume that the pattern of interactions within each scenario does not change by changing the software unit instance. (ii) We only consider the sequential execution of the software units, and we assume that the units communicate by exchanging synchronous messages. (iii) From a reliability viewpoint, we suppose that the software units are independent, namely we assume that the failure of an unit provokes the failure of the whole system. We only consider crash failures that are failures that (immediately and irreversibly) compromise the behavior of the whole system. Besides, we suppose that a unit shows the same reliability across different invocations. (iv) We assume that the operational profile of the system is the same one used for certifying the component. (v) Finally, we assume that sufficient manpower is available to independently develop in-house unit instances. Note that the above assumptions are shared with most of the models in this domain, as discussed in Sect. 14.3.

Let us suppose to be committed to assemble the system by the time T while ensuringa minimum reliability level R and spending the minimum amount of money. Let N_{ij}^{tot} be the integer variable representing the total number of tests performed on the in-house developed instance j of the ith unit.[8] Figure 14.1 summarizes the parameters and the expressions used in the model formulation. Specifically, (i) the development cost and the delivery time of an in-house instance are computed by considering the development time, the testing time and the number of tests. (ii) The reliability of the whole system can be obtained as a function of the probability of failure on demand of its elementary elements. In particular, the expression of the system reliability reported in Fig. 14.1 is the probability of a failure-free execution of the system, and hence the reliability constraint is

---

[8] The effect of testing on cost, reliability and delivery time of provided units is instead assumed to be accounted in the parameters.



1. Model Parameters

s_i	average number of invocations
μ_{ij}	probability of failure on demand of the provided instance j
c_{ij}	cost of the provided instance j
d_{ij}	delivery time of the provided instance j
τ_{ij}	average time to perform a test case on the in-house instance j
p_{ij}	probability that the in-house instance j is faulty
π_{ij}	testability of the in-house instance j
$\bar{c}_{ij}$	unitary development cost of the in-house instance j
t_{ij}	estimated development testing time of the in-house intance j
N_{ij}^{suc}	number of successful (i.e. failure-free) tests performed on the in-house j

$$N_{ij}^{suc}=(1-\pi_{ij})N_{ij}^{tot}$$

2. Cost Objective Function: $$COF=\sum_{i=1}^{n}\left(\sum_{j\in\bar{J}_i}\bar{c}_{ij}(t_{ij}+\tau_{ij}N_{ij}^{tot})x_{ij}+\sum_{j\in J_i}c_{ij}x_{ij}\right)$$

3. System Reliability : RelSyS = $$\prod_{i=1}^{n}e^{-\left(\sum_{j\in\bar{J}_i}\theta_{ij}s_ix_{ij}+\sum_{j\in J_i}\mu_{ij}s_ix_{ij}\right)}$$

4. The probability of failure on demand of the *j*-th in-house developed instance: $$\theta_{ij}=\frac{\pi_{ij}\cdot p_{ij}(1-\pi_{ij})^{N_{ij}^{suc}}}{(1-p_{ij})+p_{ij}(1-\pi_{ij})^{N_{ij}^{suc}}}$$

5. The delivery time of the software unit *i* : $$DT_i=\sum_{j\in\bar{J}_i}(t_{ij}+\tau_{ij}N_{ij}^{tot})x_{ij}+\sum_{j\in J_i}d_{ij}x_{ij}$$

Fig. 14.1 Design phase: parameters and cost, reliability and delivery time expressions

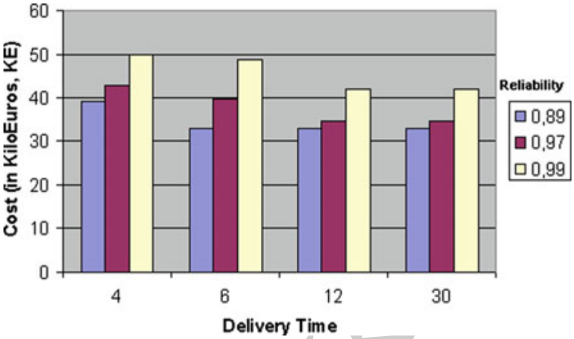
$RelSyS \geq R$. (iii) The delivery time constraints can be expressed as $DT_1 \leq T, \ldots DT_n \leq T$.

Experimenting the model on an example In order to show the practical usefulness of the model, we apply it to the example presented in Sect. 14.5.

Figure 14.2 reports a synthesis of the results obtained by solving the optimization model with different values of T and R. The former spans from 4 to 30 whereas thelatter from 0.89 to 0.99.

As expected, the total cost of the application decreases for the same value of the reliability bound R and increasing values of the delivery time limit T. On the

Fig. 14.2 Solutions for the design phase



otherhand, for the same value of T the total cost decreases while decreasing the reliability bound R (i.e. less reliable application required).

As shown in Potena et al. (xxx), the model tends to select in-house instances for increasing values of T because they become cheaper than the available provided instances. The total costdecreases while T increases because it is possible to increase the amount of testing to perform. The in-house instances remain cheaper than the corresponding provided instances even in cases where a non negligible amount of testing is necessary to make them more reliable with respect to the available provided instances.

In this example, the in-house instances result cheaper than the provided instances, but real situations may be different. In fact, an in-house unit could be built by adopting different strategies of development. Therefore, its values of cost, reliability and delivery time could vary due to the values of the development process parameters (e.g. experience and skills of the developing team). In Potena et al. (xxx) we also study the sensitivity of the model to changes in its parameters (we analyze, in particular, the behavior of the system costs at varying of non-functional requirements).

14.7 Maintenance Phase

14.7.1 After Release (Platform Independent)

In this section, we instantiate the general optimization model (3) for supporting the maintenance phase. Specifically, we show how an optimization model can support the software unit replacement maintenance activity for overcoming an *unexpected* system failure. *Unexpected* means that, on the basis of the certified reliability of the elementary software units, a failure shall not occur so early. Under the assumption that exactly one faulty software unit is present in the system, the proposed optimization model aims to maintain the system by suggesting how to reconfigure it. After a software failure occurs, our approach searches for a different system configuration (e.g. by replacing a (some) unit(s)) that minimizes the costs while raising the system reliability by a fair amount that (hopefully) allows in future to avoid unexpected failures. Indeed, the model solution may suggest either to replace a faulty software unit by a provided instance or to perform on the faulty software unit an additional number of test cases if it has been developed in-house.

The mathematical formulation, similar to that described in Sect. 14.6, has been presented in Cortellessa and Potena (2009) in the context of component-based software. In this chapter we plug the model in a general application domain, where the decisions refer to general software units rather than components.

Let S be the software architecture of a deployed system that has been assembled following the architectural approach presented in Sect. 14.6. In particular, let $(\bar{\mathrm{x}}, \mathrm{N}^{\boldsymbol{tot}})$ be the description of the instances chosen to build S at minimum cost while assuring (among others) a system reliability greater than the threshold R. For sake of readability, suppose that the possibly in-house built instance for the software unit i is included in theset J_i (and therefore $\bar{x}_{i0} = 1$ means that the ith software unit has been developed inhouse). Moreover, assume that an *unexpected* system failure occurs and that no specific monitoring action is devised to identifying the faulty unit originating the failure.

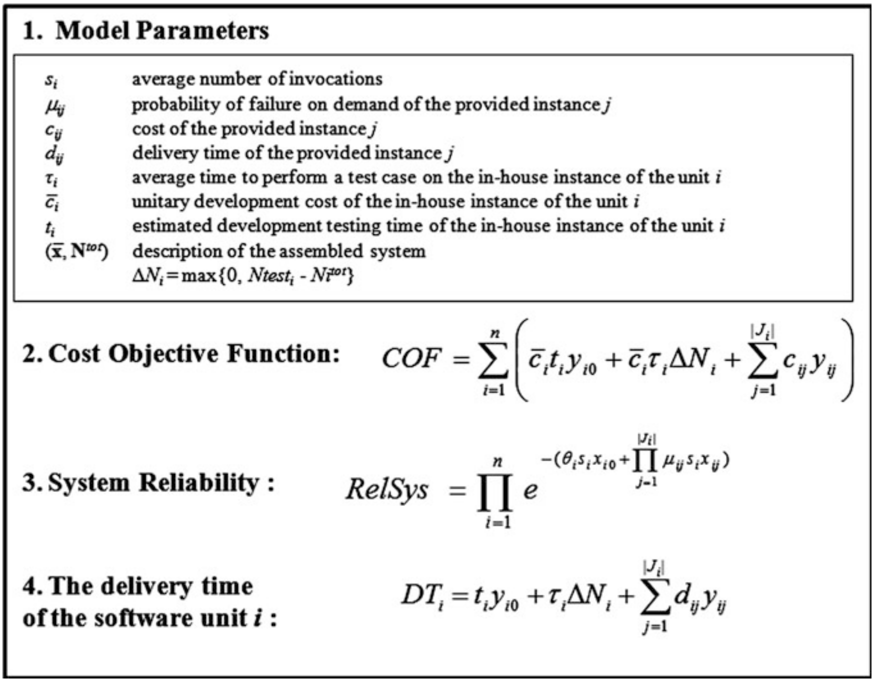
Let R' be the new reliability threshold required for the whole system (i.e. $\mathrm{R}' > \mathrm{R}$) and T' be the time limit for this maintenance action to be completed.

Given the current solution $(\bar{\mathrm{x}}, \mathrm{N}^{\boldsymbol{tot}})$, let $NTest_i (\forall i = 1, \ldots \mathrm{n})$ be the number of test cases required for the unit i in order to satisfy the new reliability threshold R'. The number ΔN_i of possible additional test cases to be performed on the ith unit is given by $\Delta N_i = max\{0, Ntest_i - N_i^{tot}\}$. Since the system has been already assembled, new costs incur only if additional tests are performed on in-house instances, i.e., $\Delta N_i > 0$, and/or existing instances are replaced by new instances bought by vendors, i.e., $\bar{x}_{ij} = 1$ and $x_{ij} = 0$. The latter case can be modeled by introducing a new binary variable $y_{ij} \geq x_{ij} - \bar{x}_{ij}$. Differently from the model presented in Sect. 14.6, the objective function and the constraints of the maintenance model take into account only such kind of costs, see Fig. 14.3.

Experimenting the model on an example In order to show the practical usefulness of the model, we apply it to the example presented in Sect. 14.5. In particular, among the results of the architectural design phase (see Sect. 14.6), we picked the system configuration $[u_{11}, u_{21}, u_{32}, (u_{40}, 128), u_{51}]$ corresponding to the case ($T = 4$, $R = 0.97$). Here, (u_{40}, 128) means that the fourth software unit has been built in-house and 128 test cases has been performed on it.

Figure 14.4 reports the results obtained from solving the optimization model for different values of T' and R'. Each bar represents the minimum cost for a given value of the delivery time bound T' and a given value of the reliability bound R'. The former spans from 5 to 50 whereas the latter from 0.98 to 0.992.

As expected, the maintenance cost of the system increases for given T' and increasing R'. However, for the same value of R' the cost decreases while increasing T' which means that a larger availability of time helps to reduce maintenance cost.



1. Model Parameters

s_i	average number of invocations
μ_{ij}	probability of failure on demand of the provided instance j
c_{ij}	cost of the provided instance j
d_{ij}	delivery time of the provided instance j
τ_i	average time to perform a test case on the in-house instance of the unit i
$\bar{c}_i$	unitary development cost of the in-house instance of the unit i
t_i	estimated development testing time of the in-house instance of the unit i
$(\bar{\mathbf{x}}, \mathbf{N}^{tot})$	description of the assembled system
	$\Delta N_i = \max\{0, Ntest_i - N_i^{tot}\}$

2. Cost Objective Function:

$$COF = \sum_{i=1}^{n} \left(\bar{c}_i t_i y_{i0} + \bar{c}_i \tau_i \Delta N_i + \sum_{j=1}^{|J_i|} c_{ij} y_{ij} \right)$$

3. System Reliability :

$$RelSys = \prod_{i=1}^{n} e^{-(\theta_i s_i x_{i0} + \prod_{j=1}^{|J_i|} \mu_{ij} s_i x_{ij})}$$

4. The delivery time of the software unit *i* :

$$DT_i = t_i y_{i0} + \tau_i \Delta N_i + \sum_{j=1}^{|J_i|} d_{ij} y_{ij}$$

Fig. 14.3 Maintenance phase: cost, reliability and delivery time expressions

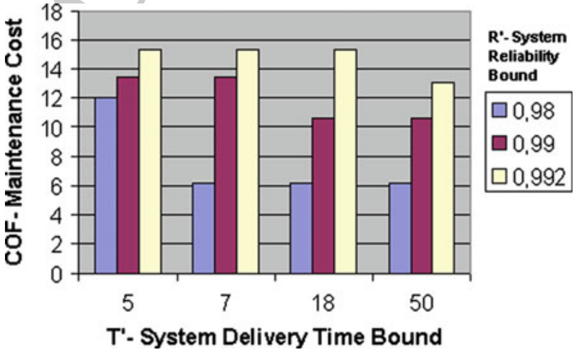


Fig. 14.4 Model solutions for the maintenance phase

The model suggests restructuring the system by working on the second and fourth software units: in some cases it suggests to perform additional testing on the fourth unit, while in all cases it argues to replace the second software unit with either its in-house instance or with its second or third provided instance available.

If we increase the value of R′ to 0.995 and set T′ = 18, then the model provides the solution $[u_{11}, u_{23}, u_{32}, (u_{40}, 452), u_{52}]$, with a maintenance cost equal to 26.268 KE and a system reliability equal to 0.996227. In this case the model suggests replacing also the fifth unit. If it would keep the first provided instance for the fifth unit (i.e. if the fifth software unit would not be replaced), the reliability constraint would be not satisfied. In fact, the system reliability would be equal to 0.992548.

In Potena et al. (xxx) we study the sensitivity of the model to changes in its parameters. We also show how, under no-monitoring assumptions and in case a monitoring action allows identifying the faulty software unit, the model can leverage the approach to overcome an *unexpected* failure of a system, see Cortellessa and Potena (2009).

14.8 Implementation/Deployment Phase

In this section, we instantiate the general optimization model (3) in order to support the activities of the implementation/development phase. In particular, we show how changes in the hardware features may affect the system quality and therefore the software decisions. As in the previous phases, the model's solution describes the instances to choose for build up a minimum cost software architecture that satisfies reliability and performance constraints. In addition, the model of the deployment phase also suggests the hardware nodes on which the software unit shall be deployed.

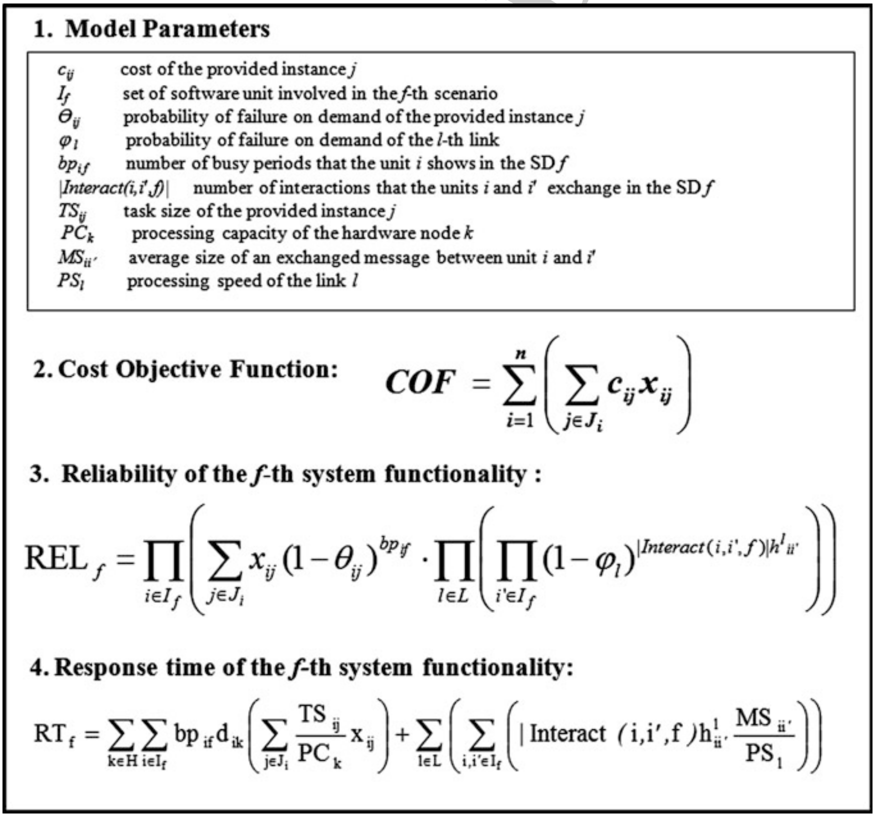
The mathematical formulation makes the following significant assumptions. (i) We assume that an UML Sequence Diagram (SD) describes the dynamic of each available functionality in terms of interactions that take place between software units (however, multiple Sequence Diagrams could be lumped by using the methodology suggested in Uchitel et al. 2003). (ii) The communication between two components co-located in the same node is assumed totally reliable, because it does not use any hardware links. (iii) Finally, we make all the assumptions of the model that we have introduced for the architectural design phase (see Sect. 14.6).

Let H be the set of hardware nodes on which the software units can be deployed, and L the set of (uni-directional) network links between hardware nodes. A link implementsa connectors between components deployed on different hardware nodes.

Additional binary variables $d_{ik}(i \in S, k \in H)$ and $h^{l}_{ii'}(l \in L, i, i' \in S)$ are needed to describe how to deploy software units on hardware nodes and how connect-software units to each other. In particular, (i) d_{ik} is equal to 1 if the node k is chosen for software unit i, and 0 otherwise, and (ii) $h^{l}_{ii'}$ is equal to 1 if the link l is chosen to connect the software units i and i′, and 0 otherwise. Each software unit i must bedeployed on exactly one node k, i.e., $\sum_{k \in H} d_{ik} = 1, \forall i \in S$, and a path must existbetween the components i and i′ if a call exists between them. The latter condition can be easily expressed as network flow constraints. Also constraints on the capacity of the nodes and the bandwidth of the network links have to be considered, see Potena et al. (xxx) for details.

Assume that the performance of the system is measured in terms of calls' response time, and that a maximum threshold $ResT$ has been given. The response time RT_f of the functionality f can be obtained as a function of the processing time and the network time, see Fig. 14.5. In a worst-case scenario, all the functionalities should satisfy the performance threshold, hence the constraints $RT_1 \leq ResT, \ldots RT_{|F|} \leq ResT$ have to be included in the formulation. Alternatively, in an average-case scenario, the response time RT of the whole system can be computed in terms of arrival rate λ_f of the callsfor the fth functionality as $RT = \sum_{f \in F} \frac{\lambda_f}{\sum_{i \in F} \lambda_i} RT_f$, and therefore the performance constraint can be simply expressed as $RT \leq ResT$.

The evaluation of the reliability of each functionality, see Fig. 14.5, takes into account that two software units may be connect from a path of more than one link. Note that the communication between two software units co-located in the same node is assumed totally reliable, because it does not use any hardware link. Again,



1. Model Parameters

c_{ij}	cost of the provided instance j
I_f	set of software unit involved in the f-th scenario
θ_{ij}	probability of failure on demand of the provided instance j
φ_l	probability of failure on demand of the l-th link
bp_{if}	number of busy periods that the unit i shows in the SD f
$\lvert Interact(i,i',f)\rvert$	number of interactions that the units i and i' exchange in the SD f
TS_{ij}	task size of the provided instance j
PC_k	processing capacity of the hardware node k
$MS_{ii'}$	average size of an exchanged message between unit i and i'
PS_l	processing speed of the link l

2. Cost Objective Function:

$$COF = \sum_{i=1}^{n} \left(\sum_{j \in J_i} c_{ij} x_{ij} \right)$$

3. Reliability of the f-th system functionality :

$$\mathrm{REL}_f = \prod_{i \in I_f} \left(\sum_{j \in J_i} x_{ij} (1-\theta_{ij})^{bp_{if}} \cdot \prod_{l \in L} \left(\prod_{i' \in I_f} (1-\varphi_l)^{|Interact(i,i',f)| h^l_{ii'}} \right) \right)$$

4. Response time of the f-th system functionality:

$$\mathrm{RT}_f = \sum_{k \in H} \sum_{i \in I_f} \mathrm{bp}_{if} \mathrm{d}_{ik} \left(\sum_{j \in J_i} \frac{\mathrm{TS}_{ij}}{\mathrm{PC}_k} \mathrm{x}_{ij} \right) + \sum_{l \in L} \left(\sum_{i,i' \in I_f} \left(|\,\mathrm{Interact}\ (i,i',f)\, \mathrm{h}^{l}_{ii'} \cdot \frac{\mathrm{MS}_{ii'}}{\mathrm{PS}_l} \right) \right)$$

Fig. 14.5 Implementation/deployment phase: cost, reliability and performance expressions in a worst-case scenario, the constraints $REL_f \geq R, (f \in F)$ must be considered, whereas in an average-case scenario, the reliability of the system is $REL = \sum_{f \in F} \frac{\lambda_f}{\sum_{i \in F} \lambda_i} REL_f$ and the reliability constraint is $REL \geq R$.

Experimenting the model on an example In this section we conclude the example presented in Sect. 14.5.

Since in general the implementation phase takes place between the architectural design and the deployment, at the deployment time no real distinction, for sake of modeling, needs to be made between in-house and provided instances. This is why the in-house instances indicated by the model solution of the architectural design phase in the scenario ($T = 30$, $R = 0.99$) are now simply considered as possible provided instances. The hardware architecture consists of three hardware nodes, see Potena et al. (xxx) for details on the model parameters.

Figure 14.6 reports the results provided by the optimization model by setting the probability of failure of the links to a value between 0.00001 and 0.0004, the processing speed of the links to 200 bits/s (measured as the average number of bits per second), the arrival rate for a service provided by the system to 1, and the reliability required to 0.97 and 0.99. Two configurations of the processing capacities of the nodes (measured as the average number of instructions per second, ips) have been considered: the first with 60, 80, and 90 ips for the first, second and third node, respectively; the second with 50, 80, and 50 ips.

As expected, for a given configuration of processing capacities of the hardware nodes and for the same value of the probability of failure of the links, the cost decreases while decreasing the reliability required for the system. On the other hand, for the same values of reliability and probability of failure of the links, the second configuration of processing capacities requires a more expensive solution.

The deployment of the software unit could change as the probability of the failure of the links varies, even when the total cost of the system remains unchanged. Indeed, in some cases it is not possible to deploy the software unit in

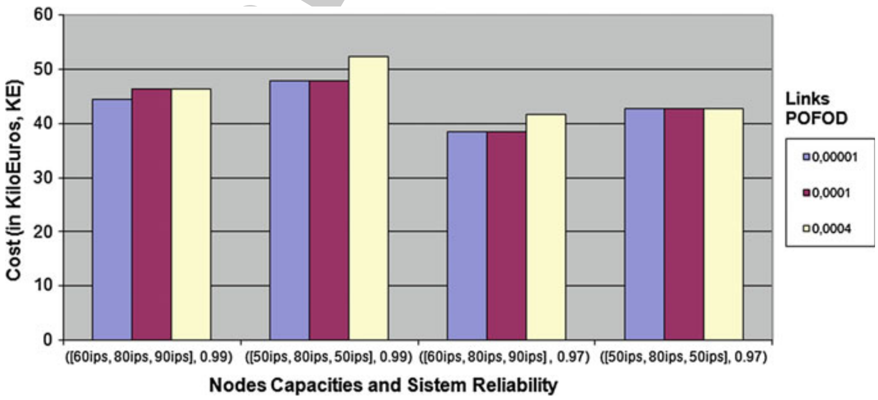


Fig. 14.6 Model solutions for the implementation/deployment phase

the same way, because this does not guarantee the reliability threshold of the system. For example, for the scenario (([60, 80, 90], 0.99), 0.0001) the model suggests a configuration of nodes that is different from the one suggested for the scenario (([60, 80, 90], 0.99), 0.0004). In fact the reliability achieved with the former configuration of nodes would be equal to 0.98853 with the probability of failure of the links fixed to 0.0004. In other cases, it is possible to deploy the software units on the hardware nodes in the same way. For example, the configuration of nodes that the model returns for the scenario (([50, 80, 50], 0.99), 0.00001) is optimal also for the scenario (([50, 80, 50], 0.99), 0.0001). In fact, the reliability achieved with the former configuration would be equal to 0.99099 with the probability of failure of the links fixed to 0.0001.

Therefore, the probability of failure of the links, that would have not emerged during the architectural design phase where the information of the links (i.e. the hardware architecture) is not taken into account (see Sect. 14.6), may sensibly affect the reliability of the system. As we have remarked in Sect. 14.1, the QoS prediction gets more accurate while progressing in the development process because more knowledge is available about the features of the system. In Potena et al. (xxx) we also study the sensitivity of the model to changes in its parameters.

14.9 Conclusions

In this chapter, we have showed how optimization models can be of support for the architectural decision-making process based on QoS tradeoffs along the whole software lifecycle. We have focused on the architectural design, the implementation/deployment, and maintenance phases, and for each phase we have introduced an optimization model that supports the decisions on the basis of the available knowledge in the specific phase. We have merged the three models in the same approach, and we have shown the usefulness of our approach by applying it to the same example in the domain of medical information systems.

The work presented in this chapter is the result of our research effort in the last years. As we report here below, besides the models formulation, we have built software tools to support the automated model generation and solution. Basing on this experience we can assert that optimization modeling is a very promising approach to formulate certain problems in the field of software quality analysis. This is especially true in cases where decisions have to be made among different alternatives that may lead to different software costs.

The most evident limitation of such approaches nowadays is the necessity to express objective functions as well as constraints in closed mathematical formulas. This is not trivial for many non-functional properties and scenarios. In addition, with the increasing complexity of software systems based on components/services, the size of these models can sensibly grow. This aspect leads to prefer heuristic search-based techniques to exact optimization tools.

Therefore we devise for the near future the necessity to work in the definition of closed mathematical formulas for different quality attributes. Beside this, we also intend to work on relaxing the model assumptions that we have introduced throughout this chapter. In particular a quite relevant aspect to work on is represented by the dependencies among different quality attributes and among parameters within the same optimization model. In this direction, we also intend to investigate the use of search based techniques, such as metaheuristics, and the multi-objective optimization for solving large scale models.

For the model for architectural design phase we have already provided the tool, called CODER (Cost Optimization under DElivery and Reliability constraints) (Cortellessa et al. 2006), which generates and solves the model automatically. We are also designing an integrated tool, based on our optimization models that may assist software designers during the whole software life cycle. It would be interesting to embed such a tool into a CASE tool, for example the one presented in Cancian et al. (2007), for supporting and automating the development of a component-based system.

Acknowledgments This work has been partially supported by the VISION European Research Council Starting Grant (ERC-240555), and by European Commission funding under the 7hFramework Programme IAPP Marie Curie program for project ICEBERG no. 324356.

References

Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., Meedeniya, I.: Software architecture optimization methods: a systematic literature review. IEEE Trans. Software Eng. **39**(5), 658–683 (2013)

Autili, M., Cortellessa, V., Ruscio, D.D., Inverardi, P., Pelliccione, P., Tivoli, M.: EAGLE: engineering software in the ubiquitous globe by leveraging uncErtainty. In: SIGSOFT FSE, pp. 488–491 (2011)

Bass, L., Klein, M., Bachmann, F: Quality attribute design primitives and the attribute driven design method. In: Software Product-Family Engineering, vol. 2290, Lecture Notes in Computer Science, pp. 169–186. Springer Berlin Heidelberg (2002)

Becha, H., Amyot, D.: Non-Functional properties in service oriented architecture – aconsumer's perspective. JSW **7**(3), 575–587 (2012)

Becker, S., Koziolek, H.: Transforming operational profiles of software components for quality of service predictions. In: Proceedings of the 10th Workshop on Component Oriented Programming (WCOP2005), 2005

Belotti, P., Lee, J., Liberti, L., Margot, F., Wächter, A.: Branching and bounds tightening techniques for non-convex minlp. Optim. Methods Softw. **24**(4–5), 597–634 (2009)

Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: overview and conceptual comparison. ACM Comput. Surv. **35**(3), 268–308 (2003)

Boehm, B., In, H.: Identifying quality-requirement conflicts. Softw. IEEE **13**(2), 25–35 (1996)

Breivold, H.P., Crnkovic, I., Larsson, M.: A systematic review of software architectureevolution research. Inf. Softw. Technol. **54**(1), 16–40 (2012)

Breivold, H.P., Larsson, M.: Component-based and service-oriented software engineering: key concepts and principles. In: EUROMICRO-SEAA, IEEE Computer Society, pp. 13–20 (2007)

Brereton, P., Budgen, D.: Component-based systems: a classification of issues. Computer **33**(11), 54–62 (2000)

Cancian, R.L., Stemmer, M.R., Schulter, A., Fröhlich, A.A.: A tool for supporting and automating the development of component-based embedded systems. J. Object Technol. **6**(9), 399–416 (2007)

Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Presti, F.L., Mirandola, R.: MOSES: A framework for QoS driven runtime adaptation of service-oriented systems. IEEE Trans. Softw. Eng. **38**(5), 1138–1159 (2012)

Chandran, S. K., Dimov, A., Punnekkat, S.: Modeling uncertainties in the estimation of software reliability. In: SSIRI, IEEE Computer Society, pp. 227–236 (2010)

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond, 2nd edn. Addison Wesley (2011)

Cortellessa, V., Marinelli, F., Potena, P.: Automated Selection of Software Components Based on Cost/Reliability Tradeoff. In: EWSA, Lecture Notes in Computer Science, vol. 4344, pp. 66–81. Springer (2006)

Cortellessa, V., Potena, P.: Path-Based error propagation analysis in composition of software services. In: Software Composition, Lecture Notes in Computer Science, vol. 4829, pp. 97–112. Springer (2007)

Cortellessa, V., Crnkovic, I., Marinelli, F., Potena, P.: Experimenting the automated selection of COTS components based on cost and system requirements. J. Univers. Comput. Sci. **14**(8), 1228–1255 (2008)

Cortellessa, V., Potena, P.: How can optimization models support the maintenance of component-based software? In: 1st International Symposium on Search Based Software Engineering, pp. 97–100 (2009)

Cortellessa, V., Mirandola, R., Potena, P.: Selecting optimal maintenance plans based on cost/reliability tradeoffs for software subject to structural and behavioral changes. In: CSMR, IEEE, pp. 21–30 (2010)

Dai, Y.-S., Xie, M., Long, Q., Ng, S.-H.: Uncertainty analysis in software reliability modeling by bayesian analysis with maximum-entropy principle. Softw. Eng. IEEE Trans. **33**(11), 781–795 (2007)

Doran, D., Tran, M., Fiondella, L., Gokhale, S.S.: Architecture-based reliability analysis with uncertain parameters. In: SEKE, pp. 629–634 (2011)

Esfahani, N., Razavi, K., Malek, S.: Dealing with uncertainty in early software architecture. In: Proceedings of ACM SIGSOFT 2012/FSE-20 (New Ideas track) (2012)

Ghezzi, C., Pinto, L., Spoletini, P., Tamburelli, G.: Managing non-functional uncertainty via model-driven adaptivity. In: Proceedings of ICSE 2013 (2013)

Gokhale, S.: Architecture-based software reliability analysis: overview and limitations. Dependable Secure Comput. IEEE Trans. **4**(1), 32–40 (2007)

Goseva-Popstojanova, K., Trivedi, K.S.: Architecture-based approach to reliability assessment of software systems. Perform. Eval. **45**(2–3), 179–204 (2001)

Goseva-Popstojanova, K., Kamavaram, S.: Software reliability estimation under uncertainty: generalization of the method of moments. In: HASE, IEEE Computer Society, pp. 209–218 (2004)

Grunske, L.: Identifying “good” architectural design alternatives with multi-objective optimization strategies. In: ICSE, ACM, pp. 849–852 (2006)

Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. **45**(1), 11:1–11:61 (2012)

http:\\www.lindo.com

Immonen, A., Niemelä, E.: Survey of reliability and availability prediction methods from the viewpoint of software architecture. Softw. Syst. Model. **7**(1), 49–65 (2008)

Jung, H.-W., Choi, B.: Optimization models for quality and cost of modular software systems. Eur. J. Oper. Res. **112**(3), 613–619 (1999)

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carrière, S.: The architecture tradeoff analysis method. In: ICECCS, pp. 68–78 (1998)

Kim, C.-K., Lee, D.H., Ko, I.-Y., Baik, J.: A Lightweight value-based software architecture evaluation. In: Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007, vol. 2, pp. 646–649, July 2007

Kontio, J.: A Case study in applying a systematic method for COTS selection. In: Proceedings of the 18th International Conference on Software Engineering, ICSE '96, IEEE Computer Society, pp. 201–209, Washington, DC, USA (1996)

Krka, I., Edwards, G., Cheung, L., Golubchik, L., Medvidovic, N.: A comprehensive exploration of challenges in architecture-based reliability estimation. In: Architecting Dependable Systems VI, vol. 5835, Lecture Notes in Computer Science, pp. 202–227 (2009)

Lee, J., Kang, S., Kim, C.-K.: Software architecture evaluation methods based on cost benefit analysis and quantitative decision making. Empir. Softw. Eng. **14**(4), 453–475 (2009)

Lundberg, L., Bosch, J., Häggander, D., Bengtsson, P.-O.: Quality attributes in software architecture design. In: Proceedings of the IASTED 3rd International Conference Software Engineering and Applications, pp. 353–362 (1999)

Malek, S., Medvidovic, N., Mikic-Rakic, M.: An Extensible framework for improving a distributed software system's deployment architecture. IEEE Trans. Softw. Eng. **38**(1), 73–100 (2012)

Meedeniya, I., Aleti, A., Grunske, L.: Architecture-driven reliability optimization with uncertain model parameters. J. Syst. Softw. **85**(10), 2340–2355 (2012)

Mirandola, R., Potena, P.: A QoS-based framework for the adaptation of service-based systems. Scalable Comput. Pract. Experience **12**(1) (2011)

Mohamed, A., Ruhe, G., Eberlein, A.: MiHOS: an approach to support handling the mismatches between system requirements and COTS products. Requir. Eng. **12**(3), 127–143 (2007)

Musa, J.: Operational profiles in software-reliability engineering. Softw. IEEE **10**(2), 14–32 (1993)

Neubauer, T., Stummer, C.: Interactive decision support for multiobjective COTS selection. In: 40th Annual Hawaii International Conference on System Sciences, 2007, HICSS 2007, pp. 283b–283b, Jan 2007

Park, J.: A high performance backoff protocol for fast execution of composite web services. Comput. Ind. Eng. **51**(1), 14–25 (2006)

Potena, P.: Optimization of adaptation plans for a service-oriented architecture with cost, reliability, availability and performance tradeoff. J. Syst. Softw. **86**(3), 624–648 (2013)

Potena, P., Crnkovic, I., Marinelli, F., Cortellessa, V.: Appendix of the chapter: software architecture quality of service analysis based on optimization models. Technical report, Dip. Informatica, Università de L'Aquila, [Online]. http://www.di.univaq.it/cortelle/docs/TRChapter.pdf **AQ2**

Roshandel, R., Medvidovic, N., Golubchik, L.: A bayesian model for predicting reliability of software systems at the architectural level. In QoSA, vol. 4880, Lecture Notes in Computer Science, pp. 108–126. Springer (2007)

Sommerville, I.: Software Engineering (7th edn.). Pearson Addison Wesley (2004)

Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley Longman Publishing Co., Inc. (2002)

Uchitel, S., Kramer, J., Magee, J.: Synthesis of behavioral models from scenarios. IEEE Trans. Softw. Eng. **29**(2), 99–115 (2003)

Vinek, E., Beran, P.P., Schikuta, E.: A dynamic multi-objective optimization framework for selecting distributed deployments in a heterogeneous environment. Procedia Comput. Sci. **4**, 166–175 (2011)

Wallnau, K., Stafford, J.A.: Dispelling the myth of component evaluation. In: Building Reliable Component-Based Software Systems (2002)

Wang, Y., Li, L., Huang, S., Chang, Q.: Reliability and covariance estimation of weighted k-out-of-n multi-state systems. Eur. J. Oper. Res. **221**(1), 138–147 (2012)

Wiesemann, W., Hochreiter, R., Kuhn, D.: A stochastic programming approach for QoS-aware service composition. In: CCGRID, pp. 226–233 (2008)

Yacoub, S., Cukic, B., Ammar, H.: A component-based approach to reliability analysis of distributed systems. In: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems, 1999, pp. 158–167 (1999)

Yang, J., Huang, G., Zhu, W., Cui, X., Mei, H.: Quality attribute tradeoff through adaptive architectures at runtime. J. Syst. Softw. **82**(2), 319–332 (2009)

Younas, M., Chao, K.-M., Laing, C.: Composition of mismatched web services in distributed service oriented design activities. Adv. Eng. Inform. **19**(2), 143–153 (2005)

Author Query Form

Book ID :	**312194_1_En**
Chapter No.:	**14**

Springer
the language of science

Please ensure you fill out your response to the queries raised below and return this form along with your corrections

Dear Author
During the process of typesetting your chapter, the following queries have arisen. Please check your typeset proof carefully against the queries listed below and mark the necessary changes either directly on the proof/online grid or in the 'Author's response' area provided below

Query Refs.	Details Required	Author's Response
AQ1	Please confirm if the section headings identified are correct.	
AQ2	Please supply year for Reference Potena et al. (xxx).	

MARKED PROOF

Please correct and return this set

Please use the proof correction marks shown below for all alterations and corrections. If you wish to return your proof by fax you should ensure that all amendments are written clearly in dark ink and are made well within the page margins.

Instruction to printer	*Textual mark*	*Marginal mark*
Leave unchanged	• • • under matter to remain	Ⓥ
Insert in text the matter indicated in the margin	ʎ	New matter followed by ʎ or ʎ(x2)
Delete	/ through single character, rule or underline or ⊢——⊣ through all characters to be deleted	ꝺ or ꝺ(x2)
Substitute character or substitute part of one or more word(s)	/ through letter or ⊢——⊣ through characters	new character / or new characters /
Change to italics	— under matter to be changed	⌄
Change to capitals	≡ under matter to be changed	≡
Change to small capitals	= under matter to be changed	=
Change to bold type	~ under matter to be changed	~
Change to bold italic	≂ under matter to be changed	≂
Change to lower case	Encircle matter to be changed	≢
Change italic to upright type	(As above)	⊹
Change bold to non-bold type	(As above)	⊹
Insert 'superior' character	/ through character or ʎ where required	Ɣ or ⅄ under character e.g. Ɣ² or ⅄²
Insert 'inferior' character	(As above)	ʎ over character e.g. ʎ₂
Insert full stop	(As above)	⊙
Insert comma	(As above)	,
Insert single quotation marks	(As above)	Ɣ‘ or ⅄‘ and/or Ɣ’ or ⅄’
Insert double quotation marks	(As above)	Ɣ“ or ⅄“ and/or Ɣ” or ⅄”
Insert hyphen	(As above)	⊢-⊣
Start new paragraph	_⌐	_⌐
No new paragraph	↩	↩
Transpose	⊔⊓	⊔⊓
Close up	linking ⁀‿ characters	⁀‿
Insert or substitute space between characters or words	/ through character or ʎ where required	Y
Reduce space between characters or words	| between characters or words affected	⊤