On the Relationships between QoS and Software Adaptability at the Architectural Level

Diego Perez-Palacin[b], Raffaela Mirandola[b], José Merseguer[a]

[a]*Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, Zaragoza, Spain*
[b]*Dip. di Elettronica e Informazione, Politecnico di Milano, Milano, Italy*

Abstract

Modern software operates in highly dynamic and often unpredictable environments that can degrade its quality of service. Therefore, it is increasingly important having systems able to adapt their behavior. However, the achievement of software adaptability can influence other software quality attributes, such as availability, performance or cost. This paper proposes an approach for analyzing tradeoffs between the system adaptability and its quality of service. The proposed approach is based on a set of metrics that allow the system adaptability evaluation.

The approach can help software architects to guide decisions on system adaptation for fulfilling system quality requirements. The application and effectiveness of the approach are illustrated through examples and a wide set of experiments carried out with a tool we have developed.

Keywords: Adaptability, Quality of Service, Software architectures

1. Introduction

In modern-day applications, software is often embedded in dynamic contexts, where requirements, environment assumptions, and usage profiles continuously change. Ergo, a key requirement for software is becoming the capability to adapt its behavior dynamically, in order to keep providing the required quality of service (QoS). As an example, consider a service-oriented application made of multiple services and components. Without adaptation, the application is prone to degrade

Email addresses: `perez.palacin@elet.polimi.it` (Diego Perez-Palacin), `mirandola@elet.polimi.it` (Raffaela Mirandola), `jmerse@unizar.es` (José Merseguer)

performance because of faulty components, messages lost between services or delays due to an increasing number of users. Using adaptation, the application can change, for example, some of the services it uses or its overall service composition [1, 2].

As an answer to this need, in recent years, industry and academia have increasingly addressed the adaptation concern, particularly with the introduction of autonomic and self-adaptive systems. General discussions concerning the issues and the state of the art in the design and implementation of self-adaptable software systems have been presented, *e.g.,* in [3, 4, 5, 6, 7, 8, 9]. These papers evidence how more and more users require that applications flexibly adapt to their contextual needs and can do so with the highest performance and availability. However, guaranteeing software adaptability can influence other quality attributes such as performance, reliability or maintainability and in the worst case, improving the adaptability of the system could decrease other quality attributes. As defended in [10], *quality attributes can never be achieved in isolation, the achievement of any one will have an effect, sometimes positive and sometimes negative, on the achievement of others.*

Finding the best balance between different, possibly conflicting quality requirements that a system has to meet and its adaptability is an ambitious and challenging goal that this research would pursue. As a first step towards this goal, this paper presents a novel approach for evaluating tradeoffs between the system adaptability and other system quality attributes, like availability or cost. The approach is based on the definition of a set of metrics that allow the evaluation of the system adaptability at the level of the architecture. This level is appropriate for dealing with software quality attributes [11, 12, 10] and several methods and tools facilitate this evaluation at architectural level [10, 13, 14, 12]. By *software architecture* we assume a set of components that make up the system. Components can require and/or offer services. Components can be in-house developed or selected from the open-world [15].

The proposed approach is useful for software architects to select from the open-world those components that can fulfill all system quality requirements. These components make up the software architecture, which will be rated according to the adaptability it shows. Evaluation will enable the tradeoff analysis of adaptability versus different software quality attributes. Far from being “a solution for every situation”, our approach can help software architects when the selected components fulfill the system requirements effectively. A software architect would apply the approach when changes in the execution context of the system occur. For example, the introduction or disposal of components; changes in the QoS of some components; or changes in the system quality requirements to fulfill.

The task of evaluating architectures is a complex one for which the software architect needs automation. To this end, we have implemented the tool SOLAR [16] (SOftware quaLities and Adaptability Relationships). The architecture of the tool and a wide experimentation are hereafter presented to support our approach.

The paper is organized as follows. Section 2 reviews the works related to our approach. Section 3 proposes metrics for quantifying software adaptability. Section 4 presents our approach for relating adaptability and a single quality attribute. Section 5 exercises the approach through an example, which involves adaptability and availability. Section 6 extends the approach to more than one quality attributes. Section 7 discusses the cases in which the approach can be used. Section 8 analyses the feasibility of the approach by means of experiments carried out by SOLAR. Finally, Section 9 concludes the paper.

2. Related Work

In the last years, as outlined in [5, 9, 8], the topic of adaptable systems has been studied in several communities and from different perspectives. Our work proposes an approach for the evaluation of the relationships between the system adaptability and its quality of service. It is based on the definition and usage of a set of metrics allowing the description and the evaluation of the system adaptability together with a formal definition of the relationships between adaptability and other quality attributes. This approach together with the provided tool can facilitate the software architects in the design reasoning process improving their abilities to deliver a satisfactory design. Therefore, hereafter, we review works appearing in the literature dealing with (i) *metrics for system adaptability*, (ii) *the trade-off analysis between different quality attributes* and (iii) *design reasoning*.

Metrics for System Adaptability. In [17] authors give a set of metrics for adaptability applicable at architectural level. The set of metrics we offer is strongly inspired by these ones. In our approach a metric not only tracks whether a requirement is adaptable or not, we also quantify *how much* adaptable it is by means of natural numbers. The same authors propose in [18] a framework, as a specialization of a general qualitative framework, to reason about non-functional requirements [19, 20]. That framework concentrates on adaptability requirements and works with quantitative values. Our work, on the contrary, is based on the addition of the adaptability property to systems in order to make them able to meet quality requirements.

In [21, 22] the authors wonder whether it is possible to measure and evaluate the adaptability of systems in order to compare different adaptive solutions. To take a step forward, they propose a set of quantitative metrics grouped by categories. These metrics are calculated statically. However, their approach can be extended to be applicable in a dynamic environment. In this direction we foresee a possible integration between our metrics definition and the approach in [21, 22]. Indeed, our approach can be used to discover architectures that can make the system able to meet the desired quality requirements. Then, we use higher-level metrics for evaluation and comparison of the already calculated suitable architectures.

In [23], the authors define a methodology for evaluating system adaptivity through a single metric. This evaluation is based on measurement traces or simulation traces that can be obtained, in test-beds, from real systems or software tools for discrete-event simulation.

Trade-off Analysis. The definition of architectural models can embody not only the software quality attributes of the resulting system, but also the trade-offs decisions taken by designers [10, 11]. The efforts to explore such trade-offs have produced the so-called scenario-based architecture analysis methods, such as SAAM and ATAM [24, 25] and others reviewed by [14]. These methods analyze the architectures with respect to multiple quality attributes exploring also trade-offs concerning software qualities in the design. The outputs of such analysis include potential risks of the architecture and the verification result of the satisfaction of quality requirements. These methods provide qualitative results and are mainly based on the experience and the skill of designers and on the collaboration with different stakeholders.

Different approaches allowing a quantitative trade-off among different software quality attributes are mainly based on the use of optimization techniques (e.g., [26, 27, 28]) or on metaheuristics approaches (e.g., [29, 30, 31]). The first ones try to find the optimal architecture by selecting the best components and taking into account possible conflicting requirements in the definition of the optimization model itself. The second type of approaches exploits evolutionary and genetic algorithms to optimize architectural models for multiple arbitrary quality attributes. A recent survey on software architecture optimization methods covering these topics has been presented in [32].

These methods, however, do not explicitly consider the adaptability as a system quality. The work more related to ours is the one presented in [33] where a trade-off analysis among quality attributes of adaptive systems is presented. This approach takes into account changes in runtime contexts and the decision to adopt an adaptation strategy is performed during runtime, when the system knows the current real context. Besides all this, our work considers trade-offs between the qualities and the adaptability.

Design Reasoning. The approach here presented can be used complementarily to other decision-making techniques to facilitate the overall design reasoning process. Several techniques exist in the literature helping software architects in this step[1]. In the following we contrast our approach with the methods closest to it: optimization problems -maximize utility-, yes/no answer to a given architecture, heuristics to find a suitable architecture.

Optimization problems (maximize utility) can find the optimal architecture given a set of requirements and their priority or utility functions; see for example [26, 28] for service-based systems . Even if some of these techniques suffer from state-space explosion, it is most likely that their execution is faster than the execution of our approach. However, for early steps of the development (stages on which our proposal is focussed) where all the system requirements are not completely stated (and the already stated requirements may change) obtaining only the current optimal architecture may be useless after some time. The results of our approach, while it does not decide for an architecture, it studies architectures properties in function of their adaptability and offers a range of possibilities to architect the system. Techniques based on yes/no answers to a given architecture regarding the analysis of its quality requirements can be found, for example, in [35, 36]. If the qualitative analysis results in a negative answer, the architect should improve the architecture and analyze it until the requirements are satisfied. This technique is well suited when the requirements cannot be properly stated as utility functions. In that case the optimization problem is useless because it could come up with an architecture that is not appropriate from the point of view of the human architect. However this technique needs manual handling for creating different architectures and large quality knowledge for changing the proper parts of the architecture that allow improving its quality. In the end, when a positive answer from the analysis is obtained, the software architect has a single solution that meets the requirements, thought it does not have information about possible

[1]Interested reader can see [34] for detailed descriptions and discussions

alternatives arising from a *what-if* analysis. Our approach, instead, offers a set of possible solutions and empowers the software architect to assess alternative architectures

For avoiding the manual modification of the architecture, heuristic techniques can be utilized. These techniques also avoid the state-space explosion problem, but they do not ensure success. These techniques automatically change the architecture to reach an architecture that: a) satisfies the quality requirements (the process stops as soon as it finds a suitable architecture) or b) stops when it is found an architecture near the optimal (no other better architecture can be found easily, e.g., with small changes in the current one) [37]. Although the current state of our approach does not consider the use of heuristics, they represent a possible extension. In the future, our approach could be improved with a good heuristic to offer the kind of results that currently offers but faster.

The principles of our approach are similar to those in [38], although the goal of the techniques diverges. In [38], authors automatically reduce the space of design choices by eliminating designs that do not satisfy some specified constraints. They do not try to find a good solution for the system design because they recognize that some system requirements cannot be specified in an analyzable formal language but they are subjective/ambiguous and they remain in designer's head; moreover, it may be better not to resolve some ambiguities until later stages in the development, when stakeholders conflicting opinions are clearer. On the contrary, their approach can automatically eliminate unfeasible system design alternatives that cannot satisfy the constraints of the subset of requirements that are formally specified, so reduce the design space for subsequent refinements in requirements or subjective design decisions. In our approach we do neither decide for an ultimate architecture but we study relationships between adaptability and quality to offer ranges of adaptability values where architectural solutions for the system reside regarding the quality requirements specified.

Summarizing, this paper proposes, with respect to existing work, the following.

- A more extensive set of architectural metrics that can be used for evaluation of the system adaptability.

- An approach that leverages these metrics for the definition of explicit relationships between adaptability and quality values, such as availability and performance. The approach is a support in the design reasoning process.

- A tool for applying the approach.

3. Architectural Adaptability Quantification

This section presents the definition of a set of metrics which quantify the potential adaptability of a software architecture. All the metrics are defined at the architectural level of a system.

3.1. Modeling Notation

For defining metrics and for evaluating quality attributes we rely on a component-and-connector view (C&C view) of the software architecture, since this view is commonly used to reason about runtime system quality attributes [39]. In C&C view *components* are principal computational elements present at runtime (e.g., COTS or in-house developed components or Internet services) [39]. Components have interfaces attached to ports. *Connectors* are pathways of interaction between components and also have interfaces or roles. The notation used in the paper for representing a C&C view is the UML component diagram. In our diagrams the components are instances and they have provided and required interfaces represented as lollipops and sockets respectively. Connectors are implicitly represented by linking the lollipop and socket of the provided and required interfaces. When the same service is required/offered by several components we join the corresponding sockets/lollipops to avoid blurring. Figure 1(b) simplifies the interfaces in (a) and also shows the implicit connectors. We omit ports but in aggregate components since they are useful to delegate interfaces, see Figure 2. From now on, we will refer interfaces also as services.

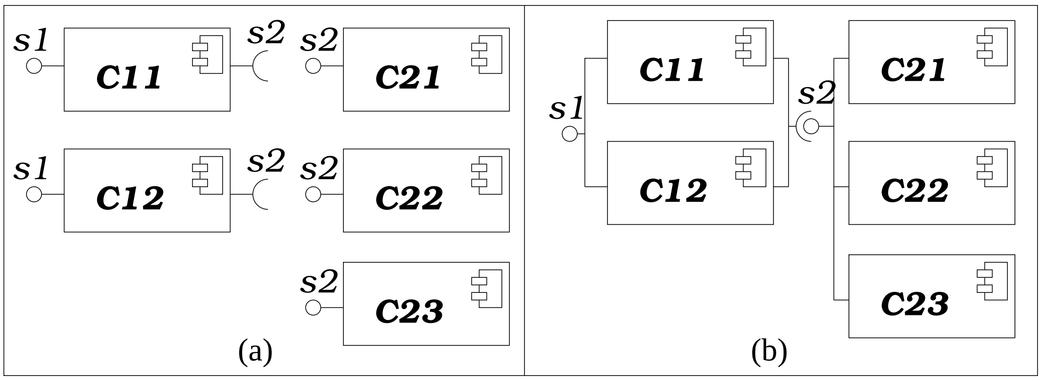


Figure 1: (a) A set of components and their interfaces, (b) The C&C view of the components in (a)

Figure 2 represents in the C&C view all the information we assumed as available to measure the adaptability of a software architecture. The information contained in the figure is: the system provides service s_1; the system architecture is made of four components, those grey shaded in the C&C view; component $C11$ provides service s_1, whereas it requires services s_2 and s_3 to accomplish its mission; s_2 is provided by $C21$ and s_3 is provided by both $C31$ and $C32$; the architect knows that there exist more components that offer services s_2 and s_3 -$C22$ and $C33$ respectively-, but s/he decided not to use them for architecting the system. For defining the metrics, we use the following formal definition of the available information: we assume the existence of n different services $s_i | i = \{1..n\}$ ($n = 3$ in Figure 2); the existence of n sets of used components in the architecture UC_i, where components in each UC_i are the ones that provide s_i ($UC_1 = \{C11\}$, $UC_2 = \{C21\}$ and $UC_3 = \{C31, C32\}$, in Figure 2); the existence of n sets of components C_i, each C_i includes the components that can provide s_i ($C_1 = UC_1$, $C_2 = UC_2 \cup \{C22\}$ and $C_3 = UC_3 \cup \{C33\}$, in Figure 2).

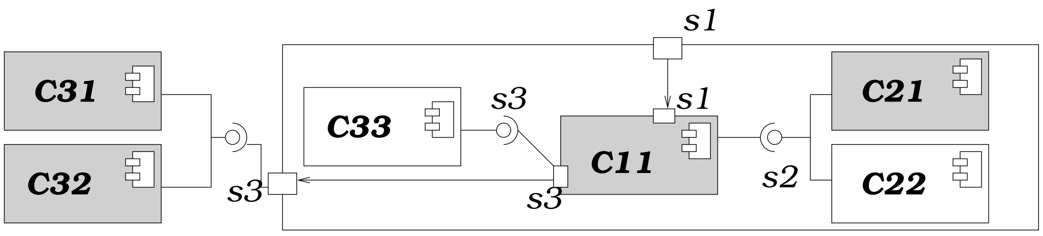


Figure 2: C&C view: discovered components and *used components* (in grey).

For the sake of simplicity, we do not represent components devoted to manage the infrastructure of the adaptive system[2]. In fact, we consider them as aggregated to the functional components, i.e., a component is supposed to add to the infrastructure a new proportional complexity for its managing. The proposed approach, indeed, concerns the assessment of trade-off between adaptability and other quality attributes. We do not explicitly deal here with the actions that lead to adaptation and that are managed by the infrastructure. Therefore, in this view the components managing the infrastructure do not influence our evaluation, but they are devoted to the implementation of the choices and to the system adaptation.

[2]Those necessary to: make requests compliant with the actual interfaces; monitor the behavior of the functional components, and; develop the logic that manages the adaptation.

3.2. Metrics

We present five metrics for measuring the adaptability of a software architecture. Four of them measure characteristics of the services of the architecture and the other measures the adaptability of the architecture as a whole.

Adaptability of the Services

Absolute adaptability of a service (AAS) measures the number of used components for providing a given service.

$$AAS \in \mathbb{N}^n \mid AAS_i = |UC_i|$$

Inspired by the *element adaptability index* in [17], which was a boolean metric (0 no adaptable, 1 adaptable), here we propose the usage of a natural number that quantifies how much adaptable a service is by counting the different alternatives to execute the service (1 no adaptable, $>$1 adaptable), where the service adaptability grows according to the number of components able to provide it.

Referring to the example in Figure 2, we observe that $AAS = [1, 1, 2]$.

Relative adaptability of a service (RAS) measures the number of used components that provide a given service with respect to the number of components actually offering such service.

$$RAS \in \mathbb{Q}^n \mid RAS_i = \frac{|UC_i|}{|C_i|}$$

It describes how each service stresses its adaptability choices and it informs how much more adaptable the service could be. RAS vector values near to one mean that the service is using almost all the adaptability potentially reachable.

Referring to the example in Figure 2, we observe that $RAS = [1, 0.5, 0.\dot{6}]$.

More adaptable architectures have vector values for RAS greater than those of less adaptable architectures. In the previous example, if we consider $UC_1 = \{C11\}, UC_2 = \{C21, C22\}$ and $UC_3 = \{C31, C32\}$, then RAS will be $[1, 1, 0.\dot{6}]$. In this new architecture, the second component of the vector is higher than that in the previous architecture, then meaning that service s_2 is more adaptable in the new architecture. Note that the maximum value for each component i of the vector is 1, meaning that the system is using all the available components to provide s_i; i.e., case in which $|C_i| = |UC_i|$.

Mean of absolute adaptability of services (MAAS) measures the mean number of used components per service.

$$MAAS \in \mathbb{Q} \mid MAAS = \frac{\Sigma_{i=1}^{n} AAS_i}{n}$$

This metric offers insights into the mean size and effort needed to manage each service.

Referring to the example in Figure 2, $MAS = \frac{4}{3} = 1.\dot{3}$.

Architectures with more adaptable services have higher values of MAAS. Besides, a $MAAS > 1$ means that the architecture includes adaptable services (at least one of the components AAS_i is greater than one). For $MAAS \leq 1$, there may be adaptable services or not (AAS should be checked in this case).

Mean of relative adaptability of services (MRAS) represents the mean of RAS.

$$MRAS \in \mathbb{Q}\{0..1\} \mid MRAS = \frac{\Sigma_{i=1}^{n} RAS_i}{n}$$

This metric informs about the mean utilization of the potential components for each service. Values of this metric range between zero and one.

Referring to the example, $MRAS = \frac{1+0.5+0.\dot{6}}{3} = 0.7\dot{2}$.

The higher the MRAS of an architecture, the more adaptable its services are, on average. The maximum value of this metric is obtained when $RAS_i = 1$ for all $i \in [1..n]$, which is in turn obtained when all services are as much adaptable as possible because they use all the available components. Therefore, a value close to one for MRAS means that, on average, services are as much adaptable as possible. A value close to zero means that: a) services can be much more adaptable (adding components not yet used), b) different architecture alternatives with the same quantity of adaptability can be created.

Adaptability of the Architecture

Level of system adaptability (LSA) measures the number of components used to make up the system with respect to the number of components that the most adaptable architecture would use.

$$LSA \in \mathbb{Q}\{0..1\} \mid LSA = \frac{\Sigma_{i=1}^{n} AAS_i}{\Sigma_{i=1}^{n} |C_i|}$$

The value of this metric ranges between zero and one. For LSA, a value of one means that the system is using all existing components for each service, i.e., $AAS_i = |C_i|$ for all $i \in \{1..n\}$, and then its adaptability is already to the maximum. A value close to one means that the market offers few choices to increase the system architectural adaptability. When a new component is bounded to the architecture, LSA increases in a constant value $(1/\Sigma_{i=1}^{n} |C_i|)$ irrespective of the number of components already considered for the same service.

Referring to the example in Figure 2, $LSA = \frac{4}{1+2+3} = 0.\dot{6}$.

Although the meaning of MRAS and LSA may seem very similar, they differ from each other in some aspects. Compared to MRAS, LSA devises a global view of the system size with respect to its maximum reachable adaptability, but does not foretell the expected value of adaptability of services (as MRSA does). To clarify the difference, consider an architecture which, for all its services s_i but one (let us call it $s_{i'}$), uses all existing components in sets C_i; while for $s_{i'}$ there is a large number of components providing it (i.e., $|C_{i'}| \gg |C_i|$) but only a few of them are used. In such case, LSA is not close to one (because a large number of components are not used, i.e., most of those in $C_{i'}$), however MRAS is close to one (in the architecture, all services but one are already using their maximum reachable adaptability; so, on average, system services are quite adaptable). Therefore, we can also see in this example that MRAS can be close to one and yet many options to increase the adaptability can exist.

Name	**Range**	**Value**	Example in Fig 2
AAS	$\mathbb{N}^n$	$\{\lvert UC_i\rvert\}$	$[1, 1, 2]$
RAS	$\mathbb{Q}^n \in \{0..1\}$	$\{\frac{\lvert UC_i\rvert}{\lvert C_i\rvert}\}$	$[1, 0.5, 0.\dot{6}]$
MAAS	$\mathbb{Q}_+$	$\frac{\Sigma_{i=1}^{n} AAS_i}{n}$	$1.\dot{3}$
MRAS	$\mathbb{Q} \in \{0..1\}$	$\frac{\Sigma_{i=1}^{n} RAS_i}{n}$	$0.7\dot{2}$
LSA	$\mathbb{Q} \in \{0..1\}$	$\frac{\Sigma_{i=1}^{n} AAS_i}{\Sigma_{i=1}^{n} \lvert C_i\rvert}$	$0.\dot{6}$

Table 1: Summary of the metrics

Table 1 summarizes the five metrics and their values for the example in Figure 2.

4. Relating Adaptability to a System Quality Attribute

As already stated in the introduction, software quality attributes can rarely be achieved in isolation. Most often, the achievement of a quality attribute has an effect, positive or negative, on the achievement of others [10]. Architectural adaptability is not an exception, and it can influence quality attributes such as performance, reliability or maintainability. In that case, an increment in the architectural adaptability can cause an improvement in some of them, but also a damage. In this section we develop an approach to study possible relationships between the architectural adaptability and the satisfiability of a given quality requirement. If such relationship exists, then architectures offering best trade-off, between adaptability and the target requirement, can be chosen. For this study we rely on the metrics presented in Section 3, which enable comparison of architectures and also the use of terms such as "adaptability increments".

Table 2 helps the understanding of the approach. In the rows we read that, when the adaptability increases then some quality attributes:

- tend to *increase* their measured values.
- tend to *decrease* their measured values.
- are not affected. We are not interested in this group since we are focussed on the influence of adaptability on the requirement.

The columns in the table consider how the quality requirement is formulated:

- as *higher than*, e.g., "system availability shall be *higher than* ..."
- as *lower than*, e.g., "system mean response time shall be *lower than*..."

	Requirement formulated as	
When adaptability increases	*Higher than*	*Lower than*
the quality attribute value increases	**Helps**	**Hurts**
the quality attribute value decreases	**Hurts**	**Helps**
the quality attribute is not affected	No effect	

Table 2: Effect of adaptability on a measured quality requirement

Each region of interest in Table 2 has been labelled as *Helps* or *Hurts* to indicate the effect of the adaptability upon the quality requirement. So, for example the first row indicates that "When the adaptability increases, if the quality attribute value increases, this *helps* to fulfill the requirements of this quality attribute formulated as *higher than*". It is worth mentioning that, as in [40], we do not intend to support the idea that a certain quality attribute always behaves the same. On the contrary, this can be only assessed after analysis, when the evolution of the measures of the quality attributes regarding the adaptability is well-known. The following examples reinforce this idea.

Example 1, given a requirement *"R: response time shall be lower than 3 sec."*, we first study in the target system whether the response time increases when a selected adaptability metric increases. In such case *R* belongs to the first row, second column, since "when adaptability increases, the response time increases and it *hurts* to fulfill *R* which has been formulated as *lower than*".

However, for another system, it may happen that *R* can be *helped* by increments in the adaptability. Even worse, for the same system, a requirement could be in *Helps* or *Hurts* depending on the system operational profile.

Example 2, consider a system that balances its workload. For high workload, the response time may decrease when the system adapts and balances its load, then the adaptability *Helps* the response time. Nevertheless, for low workloads the response time will remain about the same whether the system balances the load or not, but balancing operations will add execution overhead; so the execution time will be higher and response time can belong to *Hurts*.

A trade-off analysis between an adaptability metric and a system quality attribute can give different types of results[3]. The best case for establishing a relationship happens when results show a complete dependence between the adaptability and the selected quality attribute. In this case, we obtain graphs like the ones in Figures 3(a) and (b) (for the first and second rows in Table 2, respectively). Architects would obtain very valuable information with respect to the appropriate adaptability for the system.

However, this may be a naive hope because the component properties and their interactions may have a more profound effect into the quality attributes than the adaptability. The extreme case for this affirmation is depicted in Figure 3(c), where architectures showing very different quality attribute values can exist for any value of adaptability, meaning that the adaptability and the selected quality

[3]Our examples consider scalar metrics, for simplicity.

attribute are independent.

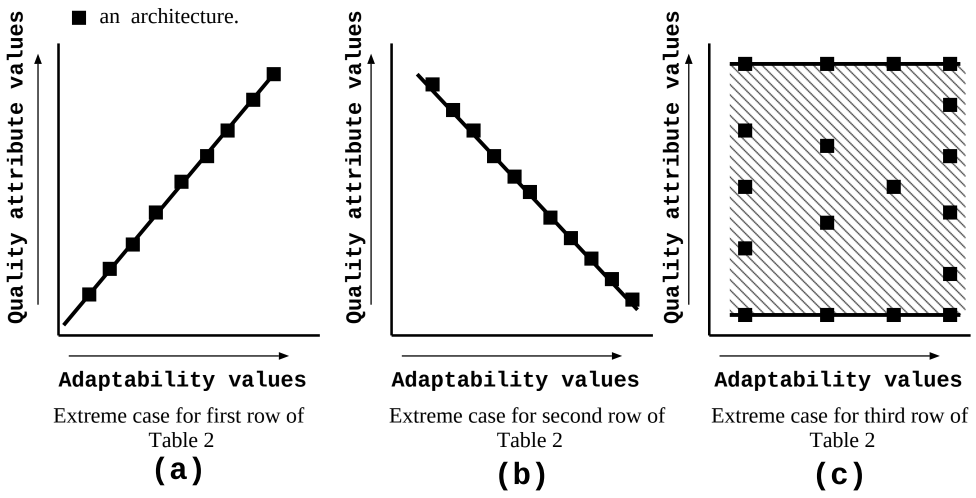


Figure 3: Extreme cases for adaptability and quality attribute relationships

Between these extreme cases we can obtain results showing some dependence. Figure 4 depicts these situations for each of the four cases in Table 2. Given a graph, the X-axis denotes increasing values A_i of an adaptability metric. Y-axis represents values for the target quality attribute. The requirement to be fulfilled is called *Requirement value*.

For each A_i, we are interested in two values: the upper bound, Q_{A_iU}, (the maximum value that the quality attribute can reach, for an architecture with adaptability A_i) and the lower bound, Q_{A_iL} (minimum value). In between these two values there exists a number of architectures that exhibit the same adaptability but different quality attribute values[4]. Putting together all Q_{A_iU} and Q_{A_iL} we obtain the graph outline. Among all Q_{A_iU} and Q_{A_iL}, we are interested in two values, $Adapt^-$ and $Adapt^+$, since they summarize the information in the graph.

For describing the meaning of $Adapt^-$ and $Adapt^+$ we focus on parts (a) and (d) in Figure 4. $Adapt^-$ is the lowest A_i for which we can find an architecture satisfying the requirement. $Adapt^+$ is the lowest A_i whose bounds, Q_{A_iU} and Q_{A_iL}, satisfy the requirement. These values indicate that to fulfill the requirement,

[4]Meaningless architectures are not considered for Q_{A_iU} and Q_{A_iL} calculation, e.g., we do not study the quality attributes of an architecture that includes a component whose provided services are not required by any other component.

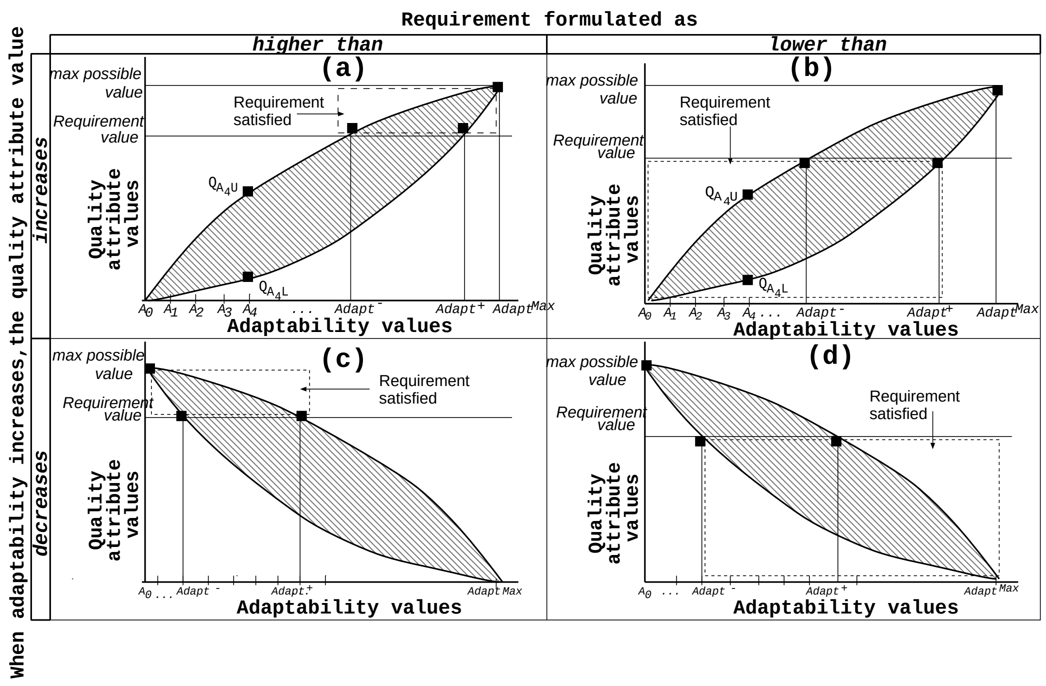


Figure 4: Relations among adaptability and other quality attributes

the architecture must have at least adaptability $Adapt^-$, and, any architecture with at least $Adapt^+$ will also satisfy it. For adaptabilities between them, there will be architectures satisfying the requirement (those highlighted in the figure) and others that will not.

In parts (b) and (c) in Figure 4 (regions where the adaptability *Hurts*), $Adapt^-$ is the threshold adaptability value for which any architecture with adaptability $A_i \leq Adapt^-$ fulfills the requirement; and $Adapt^+$ is the maximum A_i for which we know that exists some architecture that satisfies the requirement.

The four cases in Figure 4 could be merged into only two. If we negate the values of the quality attributes in the second row in Table 2, then we get the first row, so the graphs (c) and (d) of Figure 4 could be substituted for those in (a) and (b). However, it is difficult to defend that any quality attribute has this counterpart. For this reason, we prefer to consider all the four cases.

5. Example

This section presents an example to study the relation between the adaptability of a system and its availability following the method described in Section 4. The example is a web application used by students to register for an academic year in the University. The system is composed of: a presentation layer with a web GUI and mechanisms to interact with the student, and an application logic layer with the rules that approve or reject the students' proposal -a list of courses to take-. At first, students register and introduce their proposal in the system. This information is delivered to the application logic. If their proposal fulfills the University rules, this layer interacts with bank web services to proceed with the payment. Once it has finished, the control returns to the presentation layer, which sends a message and an email to the student with information about the registration process.

We assume that there exist two components that exclusively implement the application logic; a component that exclusively implements the presentation layer; a component that implements both the presentation and the application logic layers; two services for payment (two banks for paying) and three services for email sending, one of them is local to the University and two of them provided by third-parties. By availability we mean the "readiness for correct service" [41]. We assume as quality requirement to fulfill: *the system availability shall be higher than 0.9*.

Figure 5, depicts the component-and-connector view of the system. Table 3 relates components and services names to their description. Note that the view in Figure 5 slightly increments that in Figure 2 by adding service s_4 and components $C12$, $C41$ and $C42$. However, this slight increment adds a new concern because now some components offering the same service are not completely *replaceable*. For example, $C11$ cannot completely replace $C12$, since the former needs s_2 but not s_4, and the later needs the opposite.

Information required to compute availability. In Figure 5 we depict some quantitative information needed to compute the system availability. For simplicity, this information appears inside the components and in a table. However a more formal approach, like the UML-MARTE [42] standard profile, could be used.

- $P_{ij}^{s_k}$ denotes the probability of component C_{ij} to require service s_k.
- $N_{ij}^{s_k}$ denotes the number of requests to service s_k, for executions where service s_k is required by component C_{ij}.

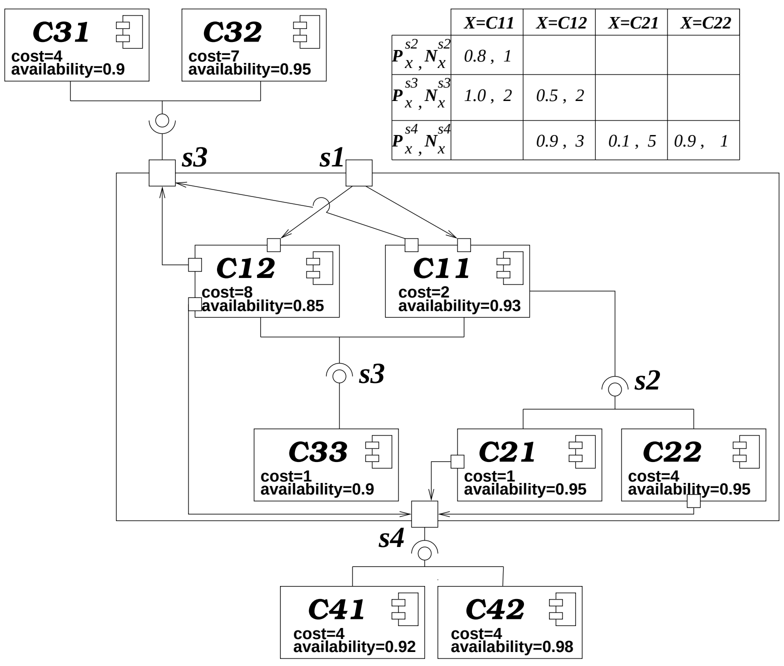
Description	**Service or Component**
Student registration	s_1
Student requirement satisfaction	s_2
Send email	s_3
Bank payment	s_4
Presentation layer	*C11*
Presentation + application logic layers	*C12*
Application logic 1	*C21*
Application logic 2	*C22*
Third-party email provider 1	*C31*
Third-party email provider 2	*C32*
Local email provider	*C33*
Bank 1 payment service	*C41*
Bank 2 payment service	*C42*

Table 3: Web application example

- The *availability* of a component is a measure obtained from the third-party provider or by monitoring the component. The *cost* of a component is an information that will be used in Section 6.
- We assume that connectors do not fail and are always available.

$P_{ij}^{s_k}$ and $N_{ij}^{s_k}$ could be combined to form the "mean number of requests per execution", however we prefer to keep them separated for the sake of system availability computation. For example, a component could be requested once per execution, another component could be requested five times per execution but only the 20% of the system executions. In both cases the "mean number of requests" is one. However, in the former case all the system executions are prone to fail, while in the latter, the remaining 80% of the executions are safe.

Availability computation. We compute availability using Markov models, for which we lean on generalized stochastic Petri nets [43]. We model a Petri net that represents the execution of the system by taking into account the availability parameter declared for components, the probability for the components to require services and the number of requests per execution for each required service. As an example, in Figure 5 service s_2 is requested by component $C11$ with probability 0.8



	X=C11	*X=C12*	*X=C21*	*X=C22*
P_X^{s2}, N_X^{s2}	0.8 , 1			
P_X^{s3}, N_X^{s3}	1.0 , 2	0.5 , 2		
P_X^{s4}, N_X^{s4}		0.9 , 3	0.1 , 5	0.9 , 1

Figure 5: C&C view of the system

and one time per request. Appendix A describes how to create the Petri net from an architectural description, and how to obtain availability results analysing the Petri net.

Results. From the metrics in Section 3 we have chosen LSA (the number of components selected to make up the system with respect to the number of components that could be used). We have chosen LSA for the sake of clarity, since scalar metrics are easier to depict than vectorial ones. For a vectorial metric it would be necessary a graph of $n + 1$ dimensions to depict the relation to availability, while relations among scalars can be represented by 2D graphs.

Following the method in Section 4 we created the corresponding graph where the quality attribute in the y-axis is availability, Figure 6 depicts it. We started considering architectures with the minimum possible LSA values, which are the architectures made of only one component (i.e., $A_0 = 1/9$). The selected component should be one of those that provide the main functionality s_1, i.e., either *C11* or *C12*. We evaluated the availability of these architectures and we obtained that: the architecture made of *C11* shows an availability equals to 0, since all of its executions require s_3, but s_3 is not provided at present; in turn, the architecture made of *C12* shows an availability equals to 0.0425. These two values are placed in the graph for x-axis equal to 1/9 and we continue generating the rest of points for the rest of adaptability values. When the graph is completed, we can see that the availability requirement belongs to *Helps*, since the availability values increase when the adaptability ones do and the requirement was formulated as *higher than 0.9*.

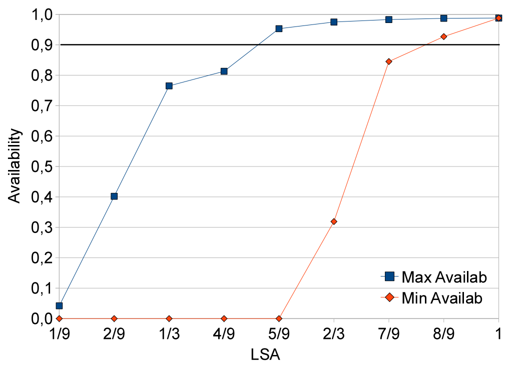


Figure 6: Relating LSA to availability

The graph shows the existence of architectures satisfying the requirement. The first solution is for an LSA equals to $\frac{5}{9}$, then $Adapt^{-} = \frac{5}{9}$. In this case, the architecture that settles the highest availability value for $LSA = \frac{5}{9}$ is made of $UC_1 = \{C11, C12\}$, $UC_2 = \{C22\}$, $UC_3 = \{C32\}$ and $UC_4 = \{C42\}$, and its calculated availability is 0.954. Regarding $Adapt^{+}$, the graph shows that all architectures with LSA $> \frac{7}{9}$ fulfill the requirement[5]. For LSA $= \frac{8}{9}$, the lower bound (worst architectural alternative) offers an availability of 0.9271. In such case, the system is made of $UC_1 = \{C11\}$, $UC_2 = \{C21, C22\}$, $UC_3 = \{C31, C32, C33\}$, and $UC_4 = \{C41, C42\}$.

Finally, we have measured using the rest of proposed metrics, for this example, the adaptability of the architectures that gave value to $Adapt^{-}$ and $Adapt^{+}$; i.e., architectures made of components $\bigcup_i UC_i = \{C11, C12, C22, C32, C42\}$ and $\bigcup_i UC_i = \{C11, C21, C22, C31, C32, C33, C41, C42\}$ for $Adapt^{-}$ and $Adapt^{+}$

[5]We remark that, following indications in Section 4, meaningless architectural alternatives have been discarded.

respectively. Table 4 extracts their measures for each metric, next we comment some of them. MRAS informs about the mean utilization of the potential components, values close to one indicate that on average the services stress their adaptability choices. Being $Adapt^{+} = 0.875$, then the system needs to stress their choices sufficiently to ensure availability. The most interesting result is for MAAS, where values greater that 1 indicate that at least one service needs adaptation; in our example, being $Adapt^{-} = 1.25$ we then know that any architecture that satisfies the availability needs adaptation.

$$n = 4 \;\; |C_1| = 2 \;\; |C_2| = 2 \;\; |C_3| = 3 \;\; |C_4| = 2$$

	$Adapt^{-}$	$Adapt^{+}$
AAS	$[2,1,1,1]$	$[1,2,3,2]$
RAS	$[1,\frac{1}{2},\frac{1}{3},\frac{1}{2}]$	$[\frac{1}{2},1,1,1]$
$MAAS$	1.25	2
$MRAS$	$0.58\dot{3}$	0.875
LSA	$0.\dot{5}$	$0.\dot{8}$

Table 4: Results for $Adapt^{-}$ and $Adapt^{+}$ for the example in Figure 5. Availability has been considered for the trade-off analysis.

6. Relating Adaptability to Several System Quality Attributes

Here we extend the approach presented in Section 4. The goal is to relate one adaptability metric to several requirements, possibly of different system quality attributes.

We started by describing the most simple case, which relates one adaptability metric to only two quality requirements. According to Table 2, each quality requirement can be expressed as “higher than” or “lower than”, and classified as *Helps* or *Hurts*. Let us consider the case in which a requirement R1 is formulated as “higher than” in *Helps*, and a requirement R2 is formulated as “lower than” in *Hurts*. So, R1 and R2 belong to the first row in Table 2. For R1 we could obtain a graph like that in Figure 4(a) and for R2 one like that in Figure 4(b), let us call them Q1 and Q2 respectively. Figure 7 depicts all possible combinations of Q1 and Q2 for the case we are analysing.

Our goal is to discover the existence of architectures that fulfill R1 and R2. To this end we only need to know the position in the X-axis for $Adapt^{+}_{Q1}$, $Adapt^{-}_{Q1}$,

$Adapt^+_{Q2}$, $Adapt^-_{Q2}$, since they determine when a requirement is fulfilled. These four values can be arranged in $4!$ different permutations. However, by definition $Adapt^-_{Q1} \leq Adapt^+_{Q1}$ and $Adapt^-_{Q2} \leq Adapt^+_{Q2}$, then the number of permutations is reduced to $\frac{4!}{2!2!} = 6$, which correspond with the six graphs in Figure 7.

It should be noted that the values in the Y-axis correspond to different scales, e.g., reliability values for R1 and cost values for R2. This means that the superposition of graphs we do in Figure 7 is artificial, so the relative positions of Q1 and Q2 are not important, the important issue is the position of the four values for $Adapt$, as stated in the previous paragraph.

Let us focus on graph (a) in Figure 7. The following cases can occur:

- We use the symbol $\forall$ to mark the region where all the architectures fulfill R1 and R2. All the architectures fulfill R1 since we have overtaken $Adapt^+_{Q1}$. All the architectures fulfill R2, since we have not reached $Adapt^-_{Q2}$.
- We use the symbol $\exists$ to mark the regions where we can find at least one architecture, for each value of A_i, that fulfills R1 and R2. To understand this let us focus on the region between $Adapt^-_{Q1}$ and $Adapt^+_{Q1}$:
  - In this region we can find architectures for all A_i, since neither $Adapt^{Max}_{Q1}$ nor $Adapt^{Max}_{Q2}$ have been reached.
  - All these architectures fulfill R2 since we have not reached $Adapt^-_{Q2}$.
  - For each A_i we can find at least one architecture that fulfills R1, since we have overtaken $Adapt^-_{Q1}$.
  - From the three statements above we can conclude that in this region all A_i has at least an architecture that fulfills R1 and R2.

  Regarding the region between $Adapt^-_{Q2}$ and $Adapt^+_{Q2}$, we can say that all architectures fulfill R1, and for each A_i we can find at least one that fulfills R2, consequently our statement also holds. Obviously this happens until we reach $Adapt^{Max}_{Q1}$ or $Adapt^{Max}_{Q2}$.
- We use the symbol $\nexists$ to mark the regions where none architecture fulfills R1 and R2. In the right hand side of $Adapt^+_{Q2}$ none architecture fulfills R2. In the left hand side of $Adapt^+_{Q1}$ none architecture fulfills R1.

We now focus on graph (b) in Figure 7 to explain a case that did not appear in (a). We use the symbol 🔍 to mark the regions where it is not possible to prove the existence or absence of architectures satisfying R1 and R2.

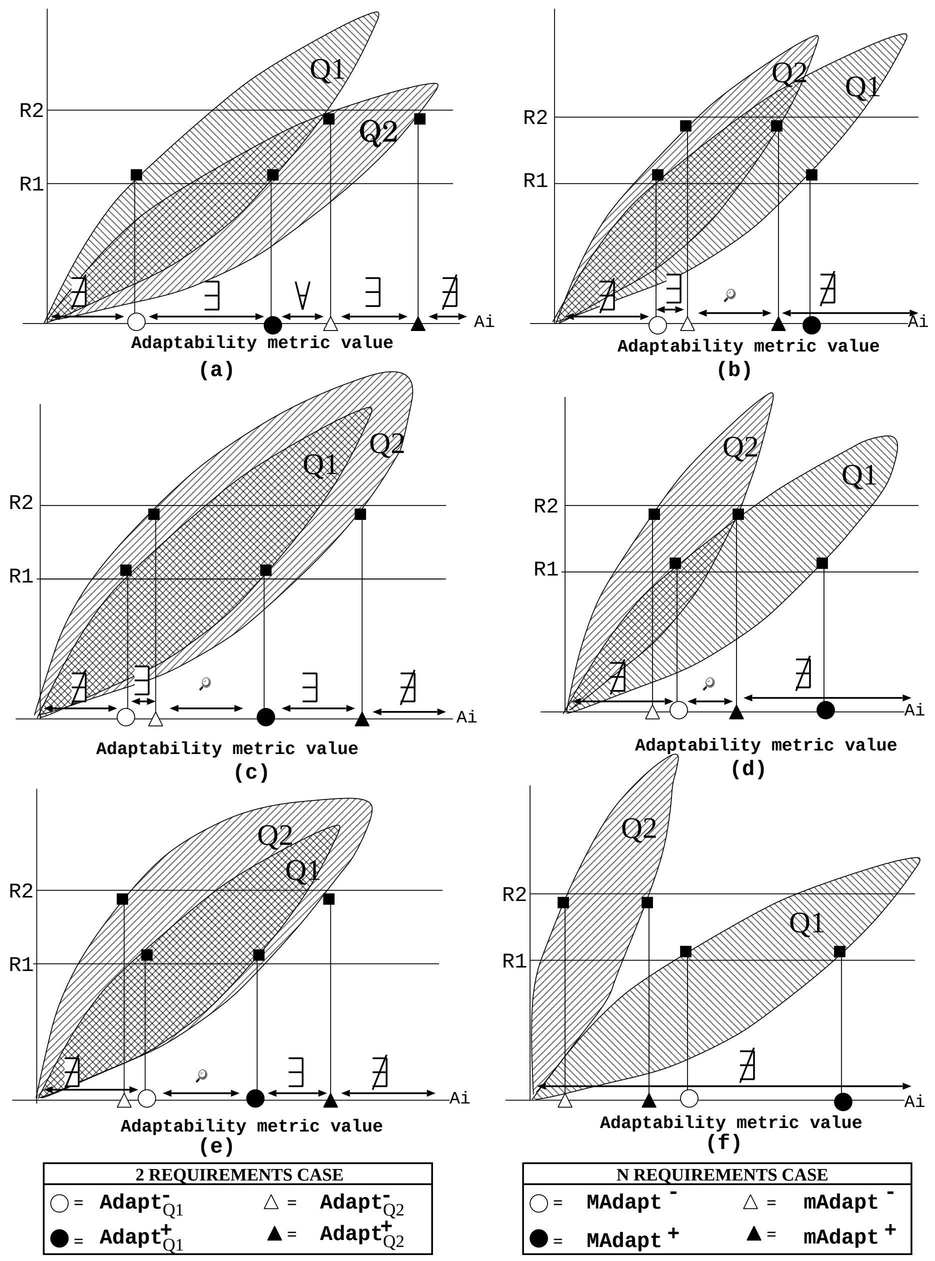


Figure 7: Relations between an adaptability metric and different quality requirements

In this case, for each A_i we can find at least one architecture that fulfills R2 since $Adapt^+_{Q2}$ has not been overtaken; but not all architectures fulfill it because $Adapt^-_{Q2}$ has been already reached. We can also find in this region, for each A_i, at least one architecture that fulfills R1 since $Adapt^-_{Q1}$ has been reached. However, we cannot affirm that one of the architectures satisfying R2 coincides with one satisfying R1. Consequently, it cannot be proved that all A_i in this interval has an architecture that fulfills R1 and R2.

The other graphs in Figure 7 have been analysed using these criteria and the results are reported in the graphs themselves using the referred symbols. The results showed by these 6 graphs are then valid to confront any two requirements belonging to the first row in Table 2 since we do not need to know the shape of the graph (only the position for $Adapt^+$ and $Adapt^-$).

For the other combinations of two requirements expressed as "higher than" or "lower than" in ***Helps*** or ***Hurts***, we could obtain graphs as those in Figure 7 and carry out the same analyses.

Finally, for the case of relating adaptability with more than two requirements, we replace the four values for comparison in the X-axis -$Adapt^-_{Q1}$, $Adapt^+_{Q1}$, $Adapt^-_{Q2}$ and $Adapt^+_{Q2}$- with $MAdapt^-$, $MAdapt^+$, $mAdapt^-$and $mAdapt^+$ respectively; which are obtained obtained as follows:
$\forall req \in Helps$, $MAdapt^- = max(Adapt^-)$ and $MAdapt^+ = max(Adapt^+)$
$\forall req \in Hurts$, $mAdapt^- = min(Adapt^-)$ and $mAdapt^+ = min(Adapt^+)$. Considering these four new values, the graphs in Figure 7 are also valid for N requirements, since the extreme values (max and min) point out to those requirements that could not be fulfilled.

6.1. Example: Relating adaptability to cost

In this study we come back to the example in Section 5. We assume known the price of each component, and we consider a new requirement to be satisfied: *the system cost shall be lower than 30 monetary units.*

Cost computation. For calculating the cost of an architecture we simply apply the formula:

$$Cost = \sum_i \sum_{\forall c_j \in UC_i} c_j.cost$$

We recognize that the method is simplistic[6], yet we consider that the focus of the work is on the relation between quality attributes and adaptability rather than on obtaining accurate values for the quality attributes themselves.

Results. We applied the method in Section 4, then computing the cost of the system for each value of LSA. The analysis showed that the cost requirement belongs to *Hurts*, since the cost increases when the adaptability does and it is required a cost *lower than* 30. Figure 8 depicts the results. It shows that it is possible to find solutions satisfying the requirement up to an LSA $= \frac{8}{9}$, so $Adapt^+ = \frac{8}{9}$. Moreover, all architectures with LSA lower than $\frac{7}{9}$ will satisfy the requirement (i.e., $Adapt^- = \frac{6}{9}$).

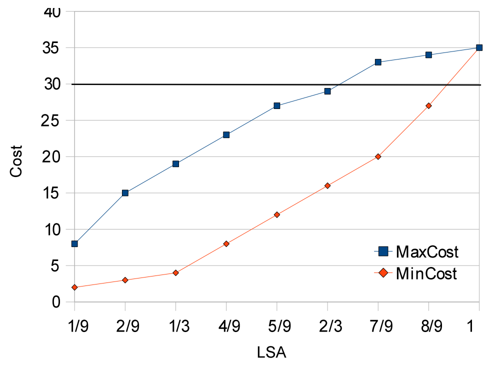


Figure 8: Relating LSA to cost

6.2. *Example: Relating adaptability to availability and cost*

Now we can practice the method developed in this section, starting from the relations between adaptability and availability -Section 5- and between adaptability and cost -Section 6.1-. Since the first study belongs to "higher than" in *Helps* and the second to "lower than" in *Hurts*, then the graphs in Figure 7 represent this case. Moreover, being $Adapt^- = \frac{5}{9}$ and $Adapt^+ = \frac{8}{9}$ for the availability study, and $Adapt^- = \frac{6}{9}$ and $Adapt^+ = \frac{8}{9}$ for the cost study, then graph (c) in Figure 7 is the one of interest. Hence, we can foretell that:

[6]We have not considered deployment costs nor advanced payment manners such as payment for execution requests, payment for temporal contract or payment for a COTS component acquisition.

- No suitable architecture can be found for an LSA $< \frac{5}{9}$ or an LSA $= 1$, since one of the requirements cannot be satisfied.

- There are suitable architectures for values of LSA $= \frac{5}{9}$, LSA $= \frac{6}{9}$ and LSA $= \frac{8}{9}$.

- There can exist suitable architectures for LSA $= \frac{7}{9}$.

7. Analysis of the Approach

This section analyses how the proposal developed in Sections 4 and 6 can assist the software architect in the design decision process. In particular, we show how the approach is useful for selecting components when changes in the environment or in the requirements occur. The goal of this analysis is to show that the range of possibilities to architect the system produced by the application of the approach meets the requirements and sometimes improves the overall system quality and/or its adaptability. This enhances the software architect reasoning abilities to both deliver a satisfactory design and improve architecture quality assurance process. As mentioned in Section 2, the time required by our approach to complete a study can be higher than the one required by other approaches concentrating on finding the architecture with the highest utility for concrete system requirements. However, the output of these studies may be useless when requirements change -fact that is pretty plausible in the early stages of system development- and the analysis has to be re-executed. Instead, using our approach, once a trade-off study has been performed, producing results like those in Figures 6 or 8, if the value of a quality requirement changes (e.g., the reliability requirement value in Figure 6), then it is not necessary to repeat the trade-off study, it is enough to redraw the asymptote of the requirement for the new value and select the new components.

In the following the analysis is guided by the four cases that lead to apply the proposal.

Definition 1. *Let us denote by:*

- C*, a set of components.*

- $ARCH$*, the set of architectures we could get by combining components, which adequately satisfy the interfaces, in* C*.*

- $Reqs$*, the set of system quality requirements.*

- $ARCH_{Reqs} \subseteq ARCH$*, the set of architectures fulfilling* $Reqs$*.*

- $Arch \in ARCH_{Reqs} \wedge Arch \subseteq C$*, the current architecture.*

- $Arch \setminus c_i$*, all components in* $Arch$ *but* c_i*.*

Case 1. The environment provides a new component c_i*.* Whether to apply the approach is a choice of the software architect, since $Arch \in ARCH_{Reqs}$. If the approach is applied, then a new set $ARCH'_{Reqs}$, where $ARCH_{Reqs} \subseteq ARCH'_{Reqs}$ is produced. A new architecture, offering better quality, in $ARCH'_{Reqs}$ could exist.

In the example in Section 5 suppose the environment provides a new component *C23 (cost=5, avail=0.98)*. Figure 9 depicts the corresponding new graphs. It is not possible finding suitable architectures for $LSA < \frac{4}{10}$ or $LSA > \frac{8}{10}$; while a solution exists for $LSA = \frac{4}{10}, \frac{5}{10}, \frac{8}{10}$; and there may exist solutions for $LSA = \frac{6}{10}, \frac{7}{10}$. Then, the new set of suitable architectures is a superset of the previous one, and the software architect has now the possibility to consider different architectures satisfying the given requirements.

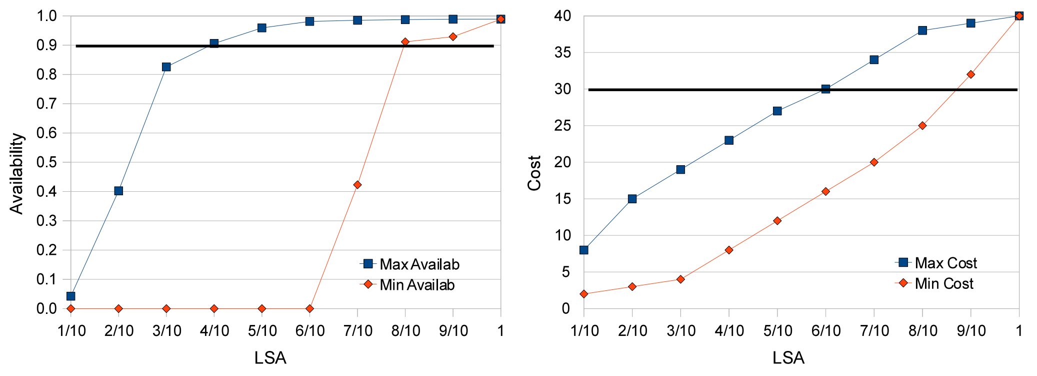


Figure 9: Same experiment as in Figure 8 but adding C23

Case 2. The environment disposes of a component c_i*.* If $c_i \notin Arch$, then obviously there is no need to apply the approach. If $c_i \in Arch$, it can happen that:

1. $Arch \setminus c_i \notin ARCH_{Reqs}$. It is mandatory for the software architect to apply the approach. Then, it could be found an $Arch' \in ARCH_{Reqs}$ or not. In the example in Section 5, $Arch$=\{*C11, C12, C21, C31, C32, C42*\} shows an availability of 0.9755 and a cost of 26mu. Now, if, for example, *C42* is disposed of, then $Arch \setminus C42$ has an availability of 0.8296 which does not meet

the requirement. If the approach is applied again, the new $Arch' =\{$*C11, C12, C21, C31, C32, C41*$\}$ is obtained with an availability equal to 0.9548 and cost of 26mu, that meets the requirements. So, the application of the approach assists the software architect in the decision process of possible architecture selection.

2. $Arch \setminus c_i$ *still meets the quality requirements*. In this case $Arch$ was over-dimensioned to fulfill the requirements. If the software architect decides to apply our approach, then a new architecture will be selected, which could be $Arch \setminus c_i$ or another one. In any case, the system quality is not deteriorated. To illustrate this case, let us consider again $Arch =\{$*C11, C12, C21, C31, C32, C42*$\}$, but now *C31* is disposed of. In this case, $Arch \setminus C31$ still meets the requirements -availability 0.953 and cost 22mu-. Therefore, it is not mandatory to apply the approach again, but the architect could apply it, since a better architecture could be found.

Case 3. An already deployed component changes some quality attribute. Some requirement has to refer to the affected quality, otherwise there is no need to apply the approach. Two cases appear:

1. c_i improves[7] some quality attribute.
   (a) $c_i \in Arch$. In this case, $Arch$ enhances this quality. The approach does not need to be applied.
   (b) $c_i \notin Arch$. The software architect should apply the approach and this may result in a new and better architecture that uses c_i. Following the example, if *C33* improves its availability up to 0.99, then a new $Arch' = \{$*C11, C12, C21, C22, C33, C42*$\}$ is obtained, which improves availability from 0.9755 to 0.98144 and cost from 26mu to 20mu.
2. c_i deteriorates some quality attribute.
   (a) $c_i \notin Arch$. $Arch$ is not affected and the approach should not be applied.
   (b) $c_i \in Arch$. The system quality is deteriorated, and the requirements are no longer met. The software architect should apply the approach.
   (c) $c_i \in Arch$. The system quality is deteriorated, but the requirements are still met. The software architect can decide to apply the approach,

[7]If the requirement is formulated as “higher than” then “improves” means that the quality value of the component increases as well. Otherwise, if it is “lower than” the quality decreases.

to see if a better architecture can be found. For example, let us consider the case where the availability of *C42* deteriorates down to 0.9. Then, the availability of $Arch$ decreases from 0.9755 to 0.94781, but the system still meets the requirements. However, the application of the approach leads to $Arch'$ $=\{$*C11, C12, C21, C32, C41, C42*$\}$ with a better availability, 0.9589, and the same cost of 26mu.

Case 4. Changes in the system quality requirements.

1. The quality requirement becomes less restrictive. The approach does not need to be applied since the requirements are obviously met. However, the the software architect can decide to apply the approach to investigate if another architecture can show a better tradeoff between qualities.
2. The quality requirement becomes more restrictive. If $Arch$ still meets the new requirement, the approach should not be applied since $Arch$ was the best choice with the same components. However, if $Arch$ does not meet the new requirement, then the approach has to be applied.

8. Experimentation and Limitations

We have developed SOLAR [16] (SOftware quaLities and Adaptability Relationships), a tool which implements the approach in Section 4. A short description of its package composition and execution behavior can be found in Appendix B. Starting from a component-and-connector view of the system SOLAR explores the design space. For each architecture SOLAR evaluates its adaptability and a target quality[8]. An *input* for SOLAR is then made of: a) a set of components, specifying for each one the four parameters described in Section 5 (*P, N, availability* and *cost*), b) the adaptability metric to evaluate, and c) the quality requirements to meet.

We judged it was important to test the performance and scalability of SOLAR, so to assess if the approach could be fast enough as to be used in real-size scenarios. Specifically, this experimentation allowed us to parametrically control the size and the structure of the generated models. In this way, we have analyzed in a controlled environment the performance of the analysis tools, getting more insights about their strengths and weaknesses. The number of possible architecture combinations for an *input* is mainly influenced by three parameters: the total

[8]SOLAR currently deals with availability and cost.

number of components in the *input*, the number of components providing each service and the number of services each component requires.

For testing the tool we have designed 681 experiments or *inputs* to be able to reason about the results. These experiments allowed managing very wide ranges for the three parameters, as follows. The parameter regarding the number components ranged from 1 to 30, we call this parameter $|C|$. For each value $|C|$, the parameter regarding the number of components providing each service ranges from the case where there is only one component that provides each service to the case where all components provide the same service. To ease the automatic generation of inputs, we considered that each service is provided by the same number of components, which means that we restricted this value to be a positive divisor of $|C|$. Since the concept "number of components providing a service" was already used in Section 3 for the cardinality of sets C_i, we call this parameter $|C_i|$. For example, for $|C| = 4$, values of $|C_i|$ are: $|C_i| = 1$ meaning that the input has four services and each service is provided by a component; $|C_i| = 2$ meaning that the input has two services and each service is provided by two components; and $|C_i| = 4$ meaning that the input has only one service and all the four components provide it. Regarding the third parameter, the number of services that each component requires, we ranged its values as much as possible for each pair of $|C|$ and $|C_i|$, i.e., from the case where each component only requires one service to the case where components require all the services but the one they offer. In other words, given $|C|$ and $|C_i|$, this parameter ranges from 1 to $\frac{|C|}{|C_i|} - 1$. We call this parameter S[9].

We defend that this set of experiments is representative. One reason is that the time required by SOLAR to execute inputs not present in the experiments, those whose component structures are less regular than those of the experiments, is upper-bounded by other regular inputs in the experiments. For example, systems in Figures 2 and 5 are simpler than the analyzed inputs where $|C| = 9$, $|C_i| = 3$, $S = 2$ and $|C| = 12$, $|C_i| = 3$, $S = 3$, respectively. Another reason is that, the execution time of SOLAR will be the same for two inputs with same component structures but different quality attribute values for their components.

Experiments results. We executed SOLAR for the 681 experiments and we recorded each execution time. We got 30 charts, one for each value of $|C|$, Figure 10 depicts five representative charts showing the execution times for the inputs whose

[9]Note that, for the particular case of inputs whose $|C| = |C_i|$, all the components offer the same service, so the number of required services is $S = 0$

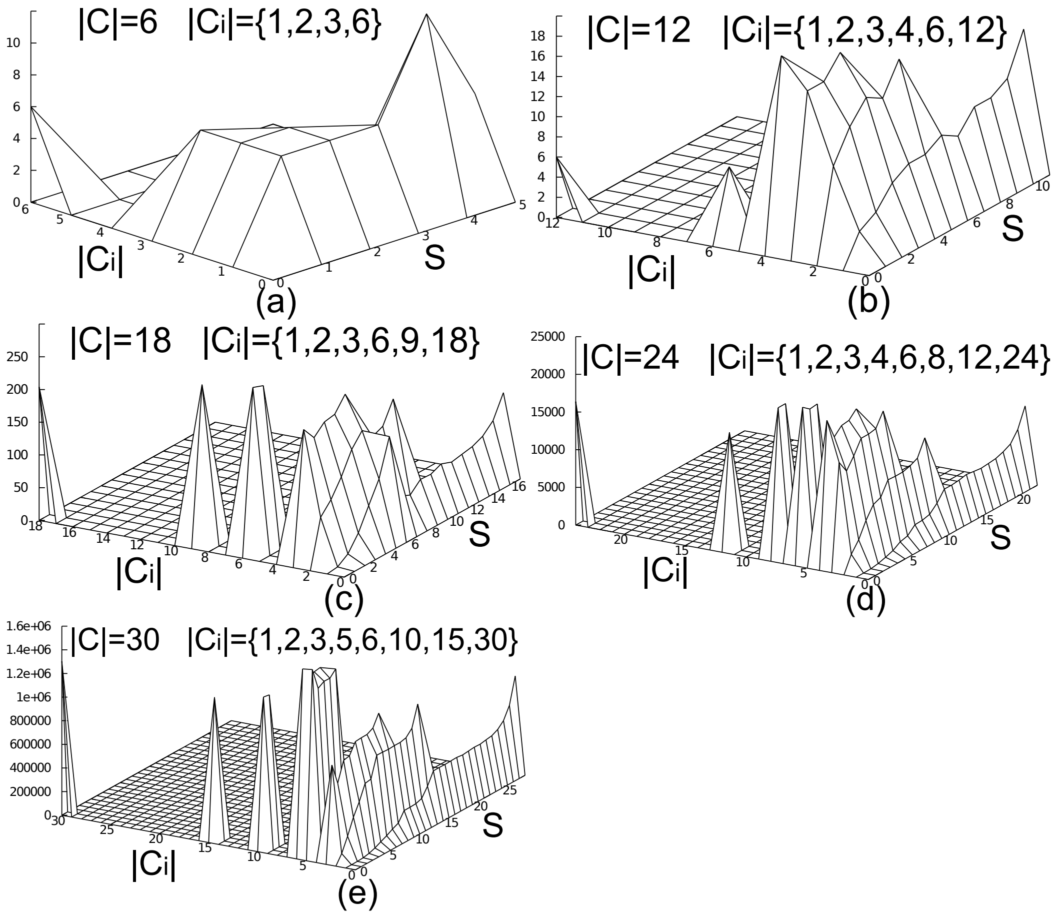


Figure 10: SOLAR execution times in milliseconds

$|C| \in \{6, 12, 18, 24, 30\}$. We can observe that the execution time strongly depends on the number of components, $|C|$, which is shown by the different time scales in the graphs. Graphs also show that execution times are also affected by the number of services provided, $|C_i|$, they follow an exponential growth for low values of $|C_i|$, but later they remain almost constant. For instance, in Figure 10(d) (where $|C| = 24$), the execution time increases exponentially up to $|C_i| = 6$ and then it remains constant for $|C_i| = 6, 8$ and $|C_i| = 12, 24$.

Considering the services required, S, we see two different cases: for low values of $|C_i|$, the execution times grow linearly with S; whereas for high values of $|C_i|$, increments in S do not affect the execution time. For instance, in Fig-

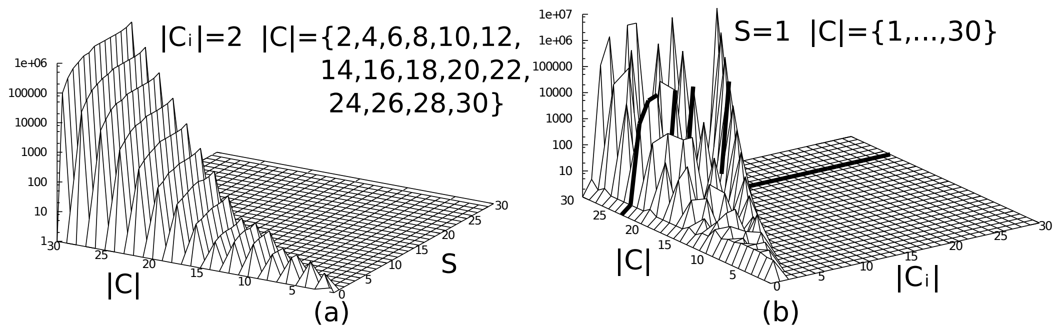


Figure 11: SOLAR execution times in milliseconds: detail of time execution with $|C|$

ure 10(d), for $|C_i| \leq 4$ the execution time increases when S does; while for $|C_i| > 4$ execution times remain constant for any value of S. The reason is that, when $|C_i|$ is small, the number of different architectures to evaluate in the *input* depends mostly on the amount of dependencies between required services of components (parameter S). However, when the number of components offering each service ($|C_i|$) grows, the number of different architectures that SOLAR has to evaluate is near the maximum even for small values of S, so increments in S do not affect the execution time in that case.

Regarding the strong dependency on the number of components $|C|$, Figures 11 (a) and (b) depict some of the experiments results in function of $|C|$. Figure 11 (a) shows that the execution time grows exponentially with $|C|$. In this figure, we fixed $|C_i|$ and we show all the results of our inputs whose $|C_i| = 2$. This figure also supports our previous explanation regarding the linear increment in the execution time with S for low values of $|C_i|$ (lines in the graph show only very slight increments because the figure is in logarithmic scale). Figure 11 (b) shows again that the execution time grows exponentially with $|C|$. In this figure, we fixed S and we show all the results of our inputs whose $S = 1$. This figure also reinforces our previous statement regarding the exponential growth of the execution time when $|C_i|$ has low value and we increment it, and the constant execution time when we increment $|C_i|$ but it has already a high value. For example, for $|C| = 24$ in Figure 11 (b) -i.e., the darkened line-, the possible values of $|C_i|$ are: 1,2,3,4,6,8,12; and we obtained that execution time of solar were: 2ms, 2.2s, 24s,19.4s, 20.5s, 19.8s and 15.4s respectively to each of the previous $|C_i|$ values.

Summary of the experiments. From the experiments we can obtain some conclusions. SOLAR takes less than one second when we evaluate inputs with less than

21 components. Most of the inputs with 21 components, some with 22 and a few with more than 22 can also be executed in less than one second. Some inputs with 30 components can last for 20 minutes. We can then say that the current prototype implementation is applicable to medium size systems. We can give more arguments in favor of our approach. Remember that Section 7 analyzed the cases when the approach should be applied, so an output of SOLAR is expected to be valid for a long time; at least until the next change in the execution context (changes in the requirements could be even more infrequent). Moreover, our experiments dealt with extreme situations, but in the real world the situation is much more relaxed: 1) not all functionalities are adaptable as in our experiments, and 2) for the ones that indeed are, there do not exist tens of alternative components offering them, on the contrary, it uses to be only a few. These reasons indicate that our approach can work in real environments even better than our analysis may reveal.

8.1. Discussion on Approach Limitations and Threats to Validity

After presenting the approach and experimenting with it, we discuss below its limitations and threats to validity.

Simple requirements. In this paper we deal with binary requirement satisfaction (i.e, satisfied or violated), but quality requirement satisfaction can often be stated in a more sophisticated continuous form in $[0..1]$. In this continuous perspective, requirement satisfaction is reinforced by a concept of architecture *utility* (for example, a requirement/s is/are satisfied with an utility value of 0.6). The proposed approach is then the basis for further enhancements based on more sophisticated requirement satisfactions, such the ones based on ranges of satisfaction. In this case, both useless (value 0) and perfect (value 1) architectures are considered together with a set of architectures with varying (increasing) utility values. In this case, it would also be possible finding other interesting adaptability values giving us information about the suitability of architectures. However, the presented approach can be hardly used when requirements satisfaction assume a continuous form where all architectures have an utility higher than 0 (useless for the requirement) and lower than 1 (perfect for the requirement). Indeed, at present the adaptability values $Adapt^+$ and $Adapt^-$ are discovered/calculated while in the continuous case these adaptability values do not exist.

Adaptability metrics. At present, regarding computation of adaptability metrics, we propose all the components to be equally important. Let's consider the case of a system in which a required service is very important and very used to accomplish system goals, while there is another one much less used and less important.

We could intuitively advice that adding one new component for providing the former service is much more important than adding one to the latter. If this information were considered for the computation of our metrics, then we would take into account the adaptability of the system weighted by the importance of the services, which would help to make our metrics more accurate. This problem could be addressed adding weight to each service. We are working indeed towards the inclusion of aspects such as the "criticality" or "importance" of the offered functionalities. The extension of the metrics to capture also the behavioral aspects is, at present, under investigation. Another direction that deserves further analysis is the integration and combination of our metrics with the ones proposed in other works (e.g., [22]) then empowering software architects to compare adaptive system designs with the system design without adaptability.

Tool performance. This first version of SOLAR can be improved to perform better. Now, for a given *input*, SOLAR first explores the design space, then it evaluates the architectures to obtain the upper and lower bounds. However, heuristics, as in [37], can be implemented to calculate the bounds. Furthermore, for some quality attributes the global maximum and minimum may depend on the local maximum and minimum; fact that can be used to avoid the current complete exploration of the design space when calculating the bounds.

Threats to Validity. Here we follow [44], where it is mentioned four kinds of threats to validity for discussion: construct, internal, conclusion and external. As concerns to *construct* and *internal* validity, our goal is on defining an approach to help architects in (automatically) finding software architectures guaranteeing adaptability and QoS tradeoffs. In this type of research a frequent problem is the lack of measures to evaluate; here we have defined them clearly. Another threat refers to how accurately the model represents the system, i.e., the "goodness" of the model. To this end we have used the C&C view, which is the common one to reason about software qualities [39]. Problems here are shared with all architectural approaches, for example, possible lack of knowledge about the real execution environment and consequently the difficulty in defining architecture parameters [39, 45]. Some methods have been defined in the literature, mainly based on estimations measuring the actual software or similar applications and also estimations from educated guesses based on experience [46, 45, 47, 48].

With respect to *conclusion* and *external* validity, instead of a real system, which is a need to support the latter, we have considered an example to show the application of the approach. However our parametric study of 681 experiments evaluated thousands of medium sized architectures and multiple combinations of provided and required services, then ranging a good number of trade-off combinations commonly present in real systems.

9. Conclusions

In this paper we have presented an approach for relating software adaptability and other quality properties. We have defined a set of metrics that quantify the software adaptability at architectural level. These metrics give means to quantitatively evaluate and compare different systems in terms of architectural adaptability and quality requirements. The approach can help software architects to find architectures satisfying all system quality requirements. The software architect applies the approach when changes in the execution context force to change the components of the architecture for satisfying quality requirements. To bring the approach to fruition we have implemented a tool that automatically performs the analysis. At present we are working towards the extension of the approach in order to overcome the presented limitations. Besides, we are still looking for a real test-bed to assess our approach in industrial settings.

Acknowledgements.

The authors would like to thank the anonymous referees for their helpful comments that definitely contribute to the improvement of the paper. This work has been partially supported by the IDEAS-ERC Project SMScom and by the CICYT - FEDER project DPI2010-20413.

References

[1] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, R. Mirandola, Qos-driven runtime adaptation of service oriented architectures, in: Proceedings of the the 7th joint meeting ESEC/FSE, ACM, New York, NY, USA, 2009, pp. 131–140. doi:http://doi.acm.org/10.1145/1595696.1595718.

[2] R. Calinescu, L. Grunske, M. Z. Kwiatkowska, R. Mirandola, G. Tamburrelli, Dynamic qos management and optimization in service-based systems, IEEE Trans. Software Eng. 37 (3) (2011) 387–409.

[3] M. C. Huebscher, J. A. McCann, A survey of autonomic computing - degrees, models, and applications, ACM Comput. Surv. 40 (3).

[4] M. Salehie, L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Trans. Auton. Adapt. Syst. 4 (2) (2009) 1–42.

[5] B. H. Cheng, al., Software engineering for self-adaptive systems: A research roadmap 5525 (2009) 1–26.

[6] J. Andersson, R. de Lemos, S. Malek, D. Weyns, Modeling dimensions of self-adaptive software systems, in: Software Engineering for Self-Adaptive Systems, Vol. 5525 of LNCS, Springer, 2009, pp. 27–47.

[7] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, A. L. Wolf, An architecture-based approach to self-adaptive software, IEEE Intelligent Systems 14 (3) (1999) 54–62.

[8] R. Calinescu, C. Ghezzi, M. Z. Kwiatkowska, R. Mirandola, Self-adaptive software needs quantitative verification at runtime, Commun. ACM 55 (9) (2012) 69–77.

[9] R. de Lemos, al., Software Engineering for Self-Adaptive Systems: A second Research Roadmap, in: Software Engineering for Self-Adaptive Systems, Vol. 7475 of LNCS, Springer, 2013, pp. 1–32.

[10] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, SEI Series in Software Engineering, Addison-Wesley, 2005.

[11] P. C. Clements, R. Kazman, M. Klein, Evaluating Software Architectures, SEI Series in Software Engineering, Addison-Wesley, 2001.

[12] C. U. Smith, L. G. Williams, Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Addison-Wesley, 2002.

[13] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni, Model-Based Performance Prediction in Software Development: A Survey, IEEE Trans. on Software Engineering 30 (5) (2004) 295–310.

[14] L. Dobrica, E. Niemela, A survey on software architecture analysis methods, IEEE Trans. on Software Engineering 28 (7) (2002) 638–653.

[15] L. Baresi, E. D. Nitto, C. Ghezzi, Toward open-world software: Issue and challenges, Computer 39 (10) (2006) 36–43. doi:http://doi.ieeecomputersociety.org/10.1109/MC.2006.362.

[16] SOLAR, `http://webdiis.unizar.es/GISED/?q=tool/solar`, Universidad de Zaragoza (2011).

[17] N. Subramanian, L. Chung, Metrics for software adaptability, in: Proc. Software Quality Management, 2001, pp. 95–108.

[18] L. Chung, N. Subramanian, Process-oriented metrics for software architecture adaptability, in: RE, IEEE Computer Society, 2001, pp. 310–311.

[19] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos, Non-Functional Requirements in Software Engineering, International Series in Software Engineering, Springer.

[20] J. Mylopoulos, L. Chung, S. Liao, H. Wang, E. Yu, Exploring alternatives during requirements analysis, IEEE Software 18 (2001) 92–96.

[21] C. Raibulet, L. Masciadri, Evaluation of dynamic adaptivity through metrics: an achievable target?, in: WICSA/ECSA, IEEE, 2009, pp. 341–344.

[22] E. Kaddoum, C. Raibulet, J.-P. Georgé, G. Picard, M.-P. Gleizes, Criteria for the evaluation of self-* systems, in: SEAMS '10: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2010, pp. 29–38.

[23] P. Reinecke, K. Wolter, A. P. A. van Moorsel, Evaluating the adaptivity of computing systems, Perform. Eval. 67 (8) (2010) 676–693.

[24] R. Kazman, L. Bass, G. Abowd, M. Webb, SAAM: A method for analyzing the properties of software architectures, in: B. Fadini (Ed.), Proceedings of the 16th International Conference on Software Engineering, IEEE Computer Society Press, Sorrento, Italy, 1994, pp. 81–90.

[25] R. Kazman, M. Klein, P. Clements, ATAM: Method for architecture evaluation, Tech. rep. (Sep. 05 2000).
URL `http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr004.pdf`

[26] D. Ardagna, R. Mirandola, Per-flow optimal service selection for web services based processes, Journal of Systems and Software 83 (8) (2010) 1512–1523.

[27] R. Calinescu, L. Grunske, M. Z. Kwiatkowska, R. Mirandola, G. Tamburrelli, Dynamic qos management and optimization in service-based systems, IEEE Trans. Software Eng. 37 (3) (2011) 387–409.

[28] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. L. Presti, R. Mirandola, MOSES: A Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems, IEEE Trans. Software Eng. 38 (5) (2012) 1138–1159.

[29] A. Aleti, S. Björnander, L. Grunske, I. Meedeniya, Archeopterix: An extendable tool for architecture optimization of AADL models, in: Proceedings of the 2009 ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES 2009), IEEE Computer Society, 2009, pp. 61–71. doi:10.1109/MOMPES.2009.5069138.

[30] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malex, J. a. P. Sousa, A framework for utility-based service oriented design in SASSY, in: Proc. of Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW), ACM, 2010, pp. 27–36.

[31] A. Martens, D. Ardagna, H. Koziolek, R. Mirandola, R. Reussner, A hybrid approach for multi-attribute qos optimisation in component based software systems, in: QoSA, Vol. 6093 of Lecture Notes in Computer Science, Springer, 2010, pp. 84–101.

[32] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, I. Meedeniya, Software architecture optimization methods: A systematic literature review, IEEE Transactions on Software Engineering 99 (PrePrints) (2012) 1. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2012.64.

[33] J. Yang, G. Huang, W. Zhu, X. Cui, H. Mei, Quality attribute tradeoff through adaptive architectures at runtime, J. Syst. Softw. 82 (2) (2009) 319–332. doi:http://dx.doi.org/10.1016/j.jss.2008.06.039.

[34] D. Falessi, G. Cantone, R. Kazman, P. Kruchten, Decision-making techniques for software architecture design: A comparative survey, ACM Comput. Surv. 43 (4) (2011) 33.

[35] V. Cortellessa, H. Singh, B. Cukic, Early reliability assessment of UML based software models, in: Proceedings of the 3rd international workshop on

Software and performance, WOSP ’02, ACM, New York, NY, USA, 2002, pp. 302–309.

[36] R. H. Reussner, H. W. Schmidt, I. H. Poernomo, Reliability prediction for component-based software architectures, Journal of Systems and Software 66 (3) (2003) 241 – 252.

[37] A. Koziolek, H. Koziolek, R. Reussner, Peropteryx: automated application of tactics in multi-objective software architecture optimization, in: 7th International Conference on the Quality of Software Architectures, QoSA 2011, 2011, pp. 33–42.

[38] A. Egyed, D. Wile, Support for managing design-time decisions, IEEE Transactions on Software Engineering 32 (5) (2006) 299–314.

[39] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford, Documenting Software Architectures: Views and Beyond, 2nd Edition, Addison-Wesley Professional, 2010.

[40] W. Hasselbring, R. Reussner, Toward trustworthy software systems, Computer 39 (4) (2006) 91–92. doi:10.1109/MC.2006.142.
URL `http://dx.doi.org/10.1109/MC.2006.142`

[41] A. Avizienis, J.-C. Laprie, B. Randell, C. E. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Sec. Comput. 1 (1) (2004) 11–33.

[42] Object Management Group, `http://www.promarte.org`, A UML Profile for MARTE. (2005).

[43] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, Modelling with Generalized Stochastic Petri Nets, J. Wiley, 1995.

[44] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2) (2009) 131–164.

[45] C. U. Smith, L. G. Williams, Performance solutions: a practical guide to creating responsive, scalable software, Addison Wesley, 2002.

[46] L. Cheung, R. Roshandel, N. Medvidovic, L. Golubchik, Early prediction of software component reliability, in: ICSE, ACM, 2008, pp. 111–120.

[47] S. S. Gokhale, K. S. Trivedi, Reliability prediction and sensitivity analysis based on software architecture, in: ISSRE, IEEE Computer Society, 2002, pp. 64–78.

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

Appendix A. Availability Computation

This appendix explains a method for creating a generalized stochastic Petri net [43] from a software architectural description. It uses the quantitative information presented in Section 5: $P_{ij}^{s_k}$ (probability of component C_{ij} to require service s_k), $N_{ij}^{s_k}$ (number of requests of component C_{ij} to service s_k) and the component availability ($C_{ij}.avalability$). We distinguish components as *terminals* and *non-terminals*. Terminals are those not needing other services (e.g., components *C31* or *C32* in Figure 5), while non-terminals do need (e.g., *C11* or *C12* in Figure 5).

Definition 2. *A terminal C_{ij} is represented as the Petri net in Figure A.12(a). Its availability is $Av(C_{ij}) = \frac{Thr(tAv)}{Thr(tAv)+Thr(tNotAv)}$, where Thr(t) is the throughput of transition t.*

The result of this quotient is always equivalent to the availability annotated in the component, according to the probability annotated in transitions `tAv` and `tNotAv` in Figure A.12(a).

Definition 3. *The availability of a* service s_i *is represented as the Petri net in Figure A.12(b).*

This Petri net models a sequential trial to execute the service in one of the components offering s_i, i.e., those in UC_i. If a component $C_{ij} \in UC_i$ is available to handle the request (there is a token in `pCijOK` place), then a token in `pSiOK` is set. On the contrary, if none of the components in UC_i is available, then a token in place `pSiFail` is set. Transition `tNoCi` sets a token in place `pSiFail` when there are no providers for s_i.

The operational profile of a service s_k requested by C_{ij} is modeled as the Petri net in Figure A.12(c). s_k is supposed to be executed with probability $P_{ij}^{s_k}$ and it is requested $N_{ij}^{s_k}$ times. If all the requests find an available provider, the execution is performed appropriately and a token is set in `pCijReqSkOK`. Otherwise, the execution cannot be completed and a token is set in `pCijReqSkFail`. Shaded

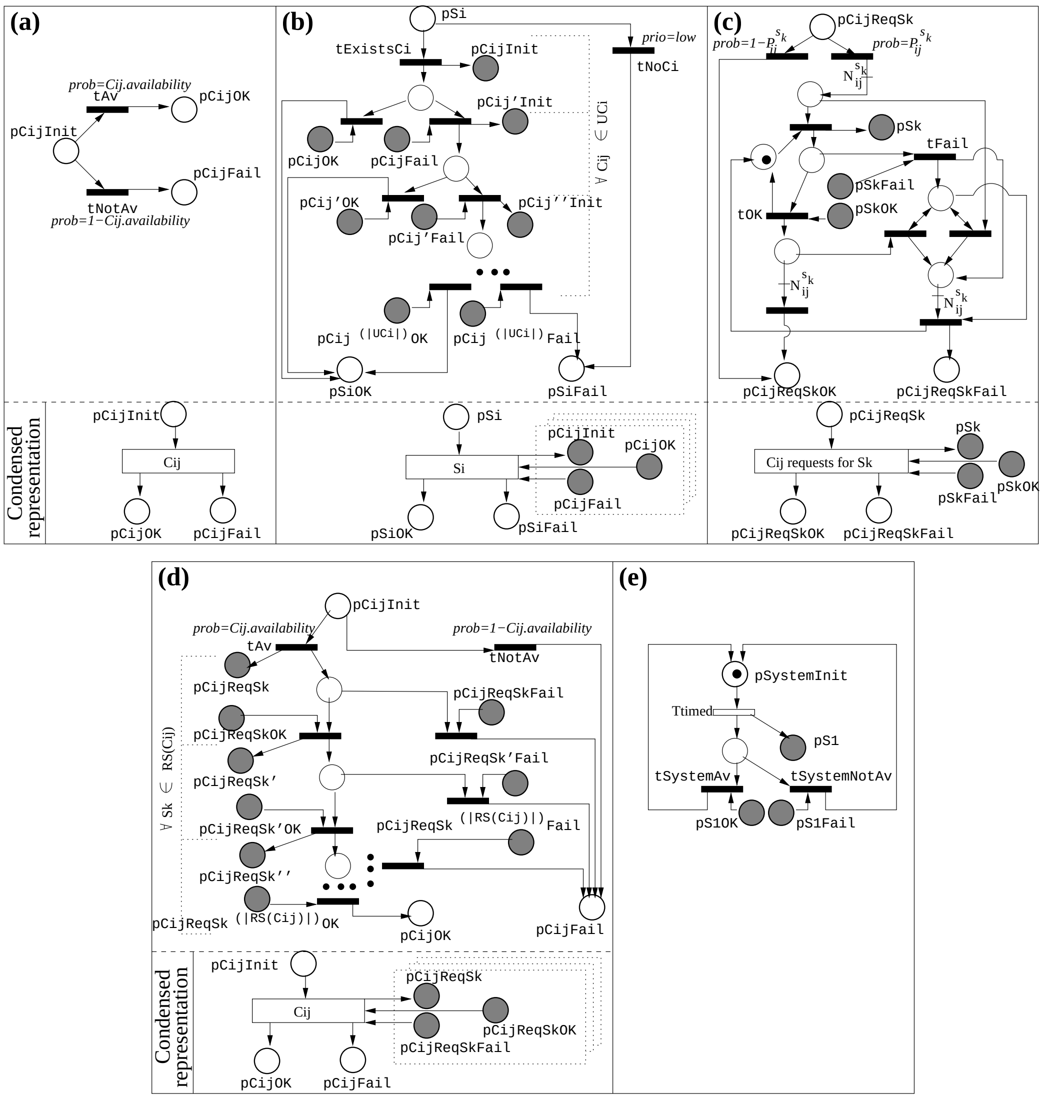


Figure A.12: Generic Petri net models for computing availability: (a) terminal component, (b) service call, (c) service requirements of a component, (d) non-terminal component, (e) system model

places `pSk`, `pSkOK` and `pSkFail` will be composed with their homonyms in part (b) of the figure.

Definition 4. *The availability of a* non-terminal C_{ij} *is represented as the Petri net in Figure A.12(d).*

If the component is not available, which happens with probability $1-C_{ij}.availability$, a token is set in `pCijFail`. If it is available, its required services are sequentially called. This behavior is modeled through groups of three shaded places per required service. These places will be composed with their homonyms in part (c) of the figure [10]. If all the service requests success, then C_{ij} is properly executed and a token is set in `pCijOK`. Otherwise, a token is set in `pCijFail`.

Definition 5. *The system availability is calculated as the quotient*

$$\frac{Thr(\texttt{tSystemAv})}{Thr(\texttt{tSystemAv}) + Thr(\texttt{tSystemNotAv})}$$

of transitions in Figure A.12(e).

This figure represents a Petri net that continuously requests for the main service s_1 of the system. Shaded places will be composed with their homonyms in part (b) of the figure. Note that $Thr(\texttt{tSystemAv}) + Thr(\texttt{tSystemNotAv})$ will be equal to `Ttimed` transition firing rate. Then, choosing a firing rate equal to 1 for `Ttimed`, system availability will correspond to $Thr(\texttt{tSystemAv})$.

The Petri net in Figure A.13 corresponds to the one of the example in Section 5, when the architecture is made of *C12*, *C31*, *C32* and *C41*. We represent the subnets in the condensed form and we only depict the shadow places. The result of the analysis unveils an availability of 0.801.

Appendix B. SOLAR Tool

Figure B.14 shows the implementation units of SOLAR, i.e., its module view. The `Quality Calculation` module owns the `QualityManager` class, which is the interface of SOLAR, and the classes for computing the system quality properties. The `Metrics` module computes the adaptability metrics as proposed in Section 3. The `CompDiag API` manages all the information in the system C&C

[10]In Figure A.12(d), we have noted as $RS(C_{ij})$ the set of services C_{ij} requires, and $|RS(C_{ij})|$ its cardinality.

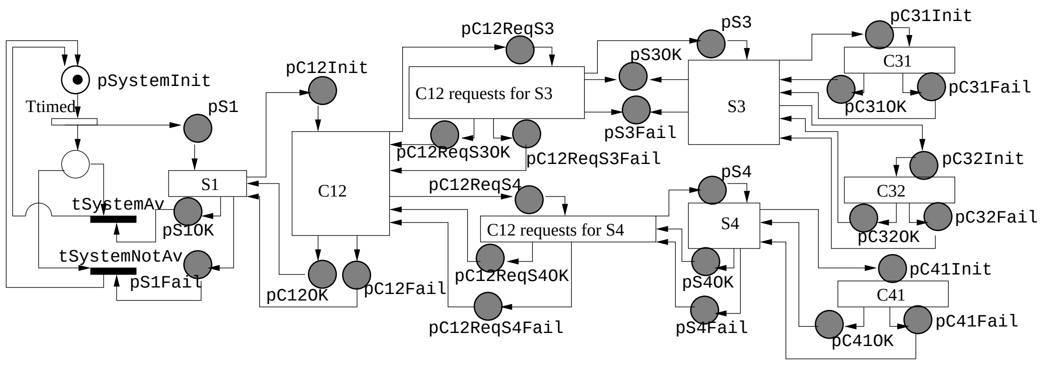


Figure A.13: Petri net for the example in Section 5 when the architecture is made of *C12*, *C31*, *C32* and *C41*.

view, i.e., the information gathered in the UML component diagram and also the parameters of the components (*P*, *N*, *availability*, *cost*). Finally, the `Parser` module, a black-box in the Figure, deals with the XML representation of the system.

When SOLAR is invoked, the `QualityManager` uses the `Parser` to get the current *input*, with the provided and required services for each component, which was stored in XML files that represent the C&C view (for instance, the input XML file for computing the studies in Sections 5 and 6 can be found in [16]). By iterating over all the possible architectures that can come from the component diagram, the `QualityManager` uses the `QualityCalculators` to compute the upper and lower quality bounds (Q_{A_iU} and Q_{A_iL}) for each adaptability value A_i. It returns these bounds and A_i values together with the architectures from which such bounds were obtained.

In the current prototype version of the tool, we have not implemented yet the translation of this C&C view of the model to the concrete language of a GSPN engine. Instead, based on general theories on how performance results are obtained from GSPNs, we have implemented in SOLAR the computation needed to get the system availability[11]. This fact, together with the fact that SOLAR has been implemented in a cross-platform language, lends the tool to be immediately tested and also easily executed with different inputs.

[11]Probability for a token to reach the place `pSystemInit` by the firing of transitions `tSystemAv` or `tSystemNotAv` in Figure A.12.

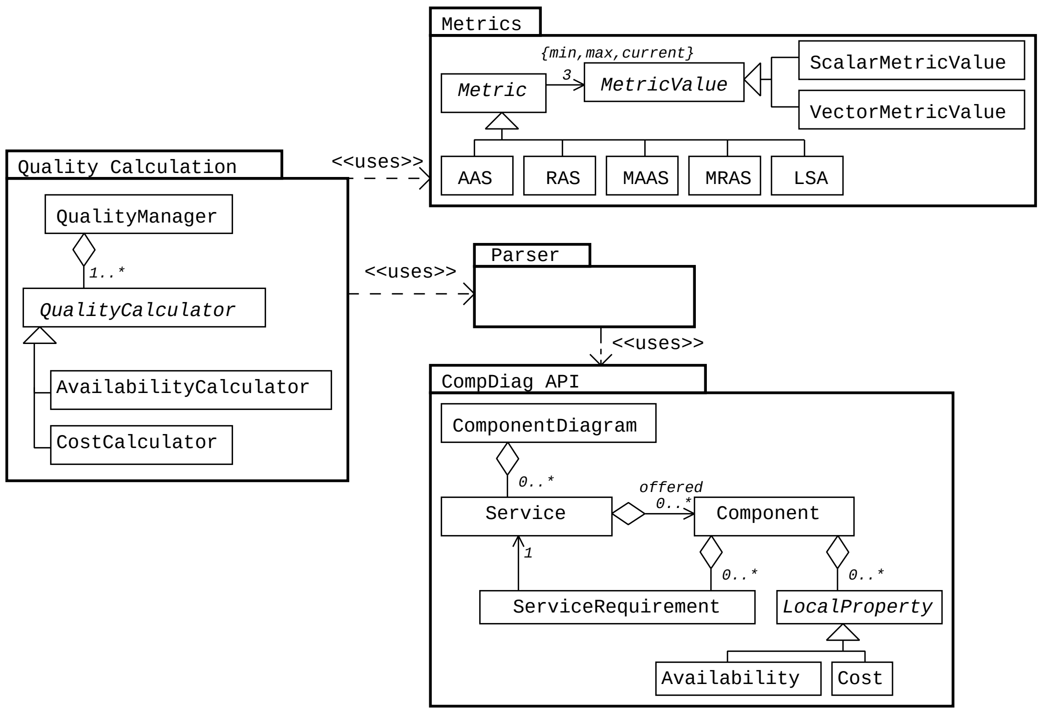


Figure B.14: SOLAR module view

User workload for using SOLAR. Currently SOLAR requires an XML file, with the information in the C&C view and with a description of the components quality attributes, that the user must provide manually. This may be a non-trivial task for non XML users. Possible improvements are: 1) In case of using SOLAR as a standalone application, a graphical user interface would ease the input generation -C&C view and quality attributes description-; 2) In case of using SOLAR as part of a software development framework, e.g., as a plug-in of the framework, the XML would be automatically generated from the architectural models created in the framework. Once obtained the XML input, no more interaction between the user and SOLAR is required, the analysis proceeds automatically.