Science of Computer Programming 80 (2014) 356–384

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Adaptation space exploration for service-oriented applications

Raffaela Mirandola [b,*], Pasqualina Potena [a], Patrizia Scandurra [a]

[a] Univ. degli Studi di Bergamo, DIMM, Dalmine (BG), Italy
[b] Politecnico di Milano, Dip. di Elettronica, Informazione e Bioingegneria, Milano, Italy

HIGHLIGHTS

- We propose a process for the adaptation of service-oriented applications.
- It is based on an optimization method.
- It is based on trade-offs between functional and extra-functional requirements.
- It uses *metaheuristic search techniques* and *functional/extra-functional patterns*.

ARTICLE INFO

Article history:
Received 15 April 2012
Received in revised form 18 September 2013
Accepted 23 September 2013
Available online 12 October 2013

Keywords:
Service-oriented applications
Software adaptation and evolution
Functional/extra-functional requirements
Optimization techniques

ABSTRACT

Service-oriented applications may require adaptation to tackle changing user needs, system intrusions or faults, changing operational environment, resource variability, etc. In order to achieve the right trade off among the functional requirements, software qualities (such as performance and reliability) and the adaptation cost itself, the adaptation decisions should involve the (*a priori*) evaluation of new alternatives to the current application design. However, the generation and evaluation of design alternatives is often time-consuming, it can be error-prone and can lead to suboptimal design decisions, especially if carried out manually by system maintainers.

This article proposes an automatic optimization process for *adaptation space exploration* of service-oriented applications based on trade-offs between functional and extra-functional requirements. The proposed method combines the use of *metaheuristic search techniques* and *functional/extra-functional patterns* (i.e., architectural design patterns and tactics). Besides, the proposed methodology relies on the standard *Service Component Architecture* (SCA) for heterogeneous service assembly and its runtime platforms. As a proof-of-concept, this article provides also an example of instantiation of the process together with an experimentation on a sample application and a numerical evaluation of the scalability of the approach.

1. Introduction

1.1. Motivation

The Service-Oriented Architecture (SOA) paradigm is widely adopted for the development of software systems through the dynamic composition of network-accessible loosely-coupled services.

In SOA, the development focus shifts from in-house custom design and implementation of system components to those activities concerning the identification, selection, and composition of services offered by third parties.

* Corresponding author.
E-mail addresses: raffaela.mirandola@polimi.it (R. Mirandola), pasqualina.potena@unibg.it (P. Potena), patrizia.scandurra@unibg.it (P. Scandurra).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.09.017

Modern service-oriented applications are 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, to be able to ensure the required functionalities in a *soft-fail* manner even in hostile or error conditions.

A second class of key requirements that today's software applications are increasingly expected to fulfill concerns extra-functional properties often specified as quality of service (QoS) constraints. As software is used more and more in business-critical and safety-critical applications, it is important to prevent the realization of software with poor QoS. Adaptation decisions should be based on the right tradeoff among these quality properties. In fact, as claimed in [1], emerging computing application paradigms require systems that are not only reliable, compact and fast, but which also optimize many different competing and conflicting objectives, like response time, throughput and consumption of resources.

In order to achieve the right trade off among the functional requirements, software qualities (such as performance and reliability) and the adaptation cost itself, the adaptation decisions should involve the evaluation of new alternatives to the current application design (e.g., by changing the selection of components, the configuration of components, the sizing, etc.). A decision, for example, taken for modifying a system functionality may be good for the satisfaction of the system reliability, but at the same time it may require a high adaptation cost for adapting the interfaces of services [2]. A major challenge is then finding the best balance among many different competing and conflicting requirements that a system has to meet and cost constraints (e.g., maximize performance and availability, while minimizing cost).

This balance might be used to answer questions such as: (i) which services/service interactions are critical to the performance and reliability of the application? (ii) how could a set of services be assembled to obtain a system, whose failure probability and response time fall under a certain upper bound (if the failure probability and response time of each services is known a priori)? (iii) how to combine architecture patterns and tactics [3,4] in order to take architectural decisions[1]?

For these multi-attribute problems, there is usually no single global solution, and the generation and evaluation of design alternatives can be error-prone and lead to suboptimal design decisions, especially if carried out manually by system architects or maintainers.

Several methods and tools have been introduced in the last years to predict and evaluate the quality of a software system (see, for example, [5]). Usually, they predict and/or analyze some quality attributes, 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 candidates that better fulfill the required quality. These evaluations can suffer of large elapsed time when the search space size increases. In such cases, the complete enumeration of possible alternatives is inefficient. To address these issues, the adoption of metaheuristic techniques has been proposed as a viable solution (e.g., see [6]).

In this article, we address 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, which is able to both fulfill the new requirements under the right trade off among the functional requirements, software qualities (such as performance and reliability) and the adaptation cost itself.

1.2. Contribution

The goal of this article is to provide an automatic optimization process for *adaptation space exploration* of service-oriented applications based on trade-offs between functional and extra-functional requirements. The optimization method combines the application of both *metaheuristic search techniques* and *architectural design patterns and tactics*.

The optimization process minimizes the adaptation costs of a system in correspondence with new requirements. These requirements induce changes in the structural and behavioral architecture of the software system. Specifically, in our process, we consider as possible changes the introduction of new functionalities (the satisfaction of new quality attributes) and the modification of the dynamics of existing functionalities (the changing of required quality thresholds). For each new requirement, we consider different sets of adaptation actions able to guarantee this new requirement. In this way, we obtain a set of decisions that lead to the definition of a new architecture, which minimizes the costs while finding the best balance among many different competing and conflicting requirements. A new architecture is, thus, obtained by modifying both its structure and its behavior.

Specifically, in order to modify the software structure, the process replaces existing software services with different available services and/or embeds new software services into the system. With respect to the changes in the system behavior, it modifies the system scenarios by removing or introducing interactions between existing services and/or between existing and new services. We adopt the OASIS/OSOA standard *Service Component Architecture* (SCA) [7] to represent service-oriented software architecture models.

This paper extends our previous work [8], in which we have presented the initial idea of an optimization process, that supports the adaptation of service-oriented applications including both static and dynamic aspects at runtime and at re-design time. In this article, we describe in detail the proposed method and we present a numerical evaluation conducted on an application example.

[1] Notice that architecture patterns are chosen in response to early design decisions, and provide the major structures in which multiple design decisions are realized. When design decisions concern quality attributes they are often called tactics. So, a tactic is a design decision whose goal is the improvement of one specific design concern of a quality attribute.

The proposed optimization approach implements and enhances the one defined in [2] by taking into account also functional aspects that allow us, among other things, to relax the assumption of independence between adaptation actions for different adaptation requirements.

Our proposed methodology does not rely on a specific adaptation phase [9]. It could be used to support both *dynamic adaptation* (or *run-time adaptation*) where adaptations may occur after deployment (meaning that the system does not need to halt) and possibly in a self-manner (self-adaptation), and *static adaptation* (or *evolution*) where changes are set before deployment or changes require some level of re-design and re-deployment to be performed.

The adoption of the proposed optimization process may require to be specialized in order to capture typical aspects of a certain phase. On one hand, the dynamic adaptation regards temporary modification, such as the re-execution of an unavailable service or a substitution of an unsuitable service, permitting to respond to changes in the requirements and/or in the application context. Therefore, in this case, in which changes may happen very often, fast approaches for the adaptation space exploration must be adopted in order to allow a prompt run-time adaptation. However, some of these changes often regard critical aspects to be applied permanently to the system, and therefore they should be merely considered as evolution steps. On the other hand, for the static adaptation, fast answers are not essential because the adaptation strategies are evaluated and carried out at re-design time or re-deployment time.

In this article, we show mainly the process adoption for the static adaptations (or evolution steps) where more sophisticated analysis are required. We postpone as future work the experimental validation of our approach also for dynamic adaptations (like dynamic service selection) and the verification of the time constraints related to the execution time of the optimization process itself.

1.3. Organization

The remainder of this article is organized as follows. Section 2 briefly surveys related works. The proposed methodology for adaptation space exploration is presented in Section 3. The running case study is presented in Section 4 together with the model parameters used for the experimental results. Possible adaptations evaluated through the application of our methodology to the running case study are presented in Sections 5.1 and 6, respectively. Section 7 presents the results of an extensive numerical evaluation conducted to analyze our optimization process sensitivity to changes in input parameters and in the component set size. Section 8 draws some conclusions and provides pointers to on-going work. An Appendix A is also included describing the selection of tactics used in the case study.

2. Related approaches

Several research efforts have been devoted in the last years to the definition of approaches and frameworks for adaptation (see, e.g., the survey [10]). Most of them typically adapt a system by adopting different service selection policies, varying system parametrization or exploiting the inherent redundancy of the Service-Oriented Architecture (SOA) environment.

For a given application domain, the behavior of an adaptation approach mainly depends on its use for evolution (at re-design time) or self-adaptation (at run-time). As already stated, our approach is not tied to any particular adaptation phase of the software life-cycle, but in this article we show mainly the process adoption for the static adaptations, where more sophisticated analysis are required. Specifically, we show the use of tactics, which typically involve substantial architectural changes. As far as dynamic environments is concerned, requiring a continuous adaptation of the composition of services, only temporary modification (such as the re-execution of an unavailable service or a substitution of an unsuitable service) should be supported. To this extend, fast approaches (see, for example, [11] and [12] detailed below), combined with techniques for estimation of quality at runtime (see Section 3.2), must be adopted in order to allow a prompt run-time adaptation used for our adaptation process.

In [11] two heuristics methods are designed for the dynamics service composition in the sensor network environment. In particular, the service-oriented domain is exploited for representing sensor network applications, and the sensor service composition process is formulated as a cost-optimization problem.

In [12] the Ra_4C (Regression-based Approach for Composition) for automated semantic web service composition at functional level is presented. Specifically, a general framework is designed where web services are specified by means of input and output parameters together with preconditions and effects. Moreover, in order to perform the web service composition, the semantic link matrix is introduced as an innovative and formal model to apply problem-solving techniques such as regression (or progression)-based search.

Here we shortly review the main methods for adaptation space exploration with the goal of improving system quality.

Rule-based Approaches, for example, include the work of Xu et al. [13] presenting a semi-automated approach to find configuration and design improvement on the model level. Parsons et al. [14] present a framework for detecting performance anti-patterns in Java EE architectures. The method requires an implementation of a component-based system, which can be monitored for performance properties. Cortellessa et al. [15] propose an approach for automated feedback generation for software performance analysis, which aims at systematically evaluating performance prediction results using step-wise refinement and the detection of performance problem patterns. All rule-based approaches share a common limitation. The model can only be changed as defined by the improvement rules. However, especially performance is a complex and cross-cutting quality criterion. Thus, optimal solutions could lie on search paths not accessible by rules.

Metaheuristic based Approaches: Evolutionary approaches are applied in Grunske [16] for the improvement of availability and costs, in [17] for the study of the trade-off between performance and cost. The usage of evolutionary algorithms together with architectural patterns is implemented in the SASSY framework for generating service-oriented architectures based on quality requirements [18]. In these approaches the derived optimization process is time-consuming.

Software architecture analysis approaches such as SAAM and ATAM [19] and others reviewed in [20], analyze the software architecture with respect to multiple quality attributes exploring also tradeoffs 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. Methods using architectural patterns and tactics for quality attributes are presented in [3,21], while [22,4] describe how software adaptation patterns can be used to help with the adaptation of service-oriented software systems after original deployment. These methods provide qualitative results and are mainly based on the experience and the skill of designers and on the collaboration with different stakeholders.

With respect to the state-of-art, the novelty of our approach can be summarized as follows.

- This is one of the few papers (to the best of our knowledge our previous work is the first one [8]) proposing an automatic optimization process for adaptation space exploration of service-oriented applications, supporting both static and dynamic aspects, based on trade-offs between functional and extra-functional requirements. This paper is an extended version of [8]. Here we describe in detail the proposed method and we present a numerical evaluation conducted on an application example.
- Our approach is not tied to any particular adaptation phase, even if the proposed optimization process may require to be specialized in order to capture typical aspects of a certain phase. Our optimization process uses a mixed approach of metaheuristic search techniques and of application of recurring (repository-based) software design solutions. Besides, this methodology relies on a formal and executable service-oriented component model called SCA-ASM [23] both for the specification and analysis of functional and non-functional aspects.
- The proposed process takes into consideration the behavior of the “components” and the “architecture” of the application. Therefore, the adaptation is driven by the analysis to characterize the system qualities (e.g., performance and reliability) behavior of the software application. If the software application is to be adapted from a collection of basic elements (e.g., components or services), then giving an answer to such questions can help to make decisions, such as which services should be picked off the shelf, and which services should be embedded into the system.
- Our framework can facilitate the work of system architects and/or maintainers. In fact, a user does not have to insert as input value architectures satisfying all changes required, but possible sets of adaptation actions (called adaptation plans) for each new requirement (see Section 3.2), which he/she want to implement.

3. Adaptation methodology

This section presents background notions on the language adopted to describe service-oriented software architectures (including the adaptation mechanism supported by the formalism) and describes the proposed *adaptation space exploration* process.

3.1. Service architecture description language and adaptation

We adopt the OASIS/OSOA standard *Service Component Architecture* (SCA) [7] to represent service-oriented software architecture models. SCA defines how heterogeneous services can be described, assembled, and deployed in a graphical (through a visual notation developed with the Eclipse-EMF environment) and XML meta-data driven way, independently of implementation languages and deployment platforms. SCA runtime environments (like Apache Tuscany [24] and FraSCAti [25], to name a few) enable then developers to execute/debug the resulting composite application. By using the SCA programming model, a service-oriented application can be built or adapted quickly and effectively, separating the business logic from technology concerns and enabling re-use of existing components. Additionally, by exploiting the SCA Policy Framework [7], the designer can specify for components necessary meta-data annotations. These are useful for providing metrics that can be extracted from the model for non-functional analysis purposes, and for representing policies that can be guaranteed by the runtime platforms.

Fig. 1 shows an *SCA assembly* of a service-oriented application as a composition of SCA components using the SCA graphical notation. An *SCA component* is a piece of software that has been configured to provide its business functions (operations) for interaction with the outside world. This interaction is accomplished through: *services* that are externally visible functions provided by the component; *references* (access points to services) wired to services provided by other components; *properties* allowing for the configuration of a component implementation with externally set data values; and *bindings* that specify access mechanisms used by services and references according to some technology/protocol (e.g., WSDL binding to consume/expose web services, JMS binding to receive/send Java Message Service, etc.). Services and references are typed by *interfaces*. An interface describes a set of related operations (or business functions) which as a whole make up the service offered or required by a component. The provider may respond to the requester client of an operation invocation with zero or more messages. These messages may be returned synchronously or asynchronously.

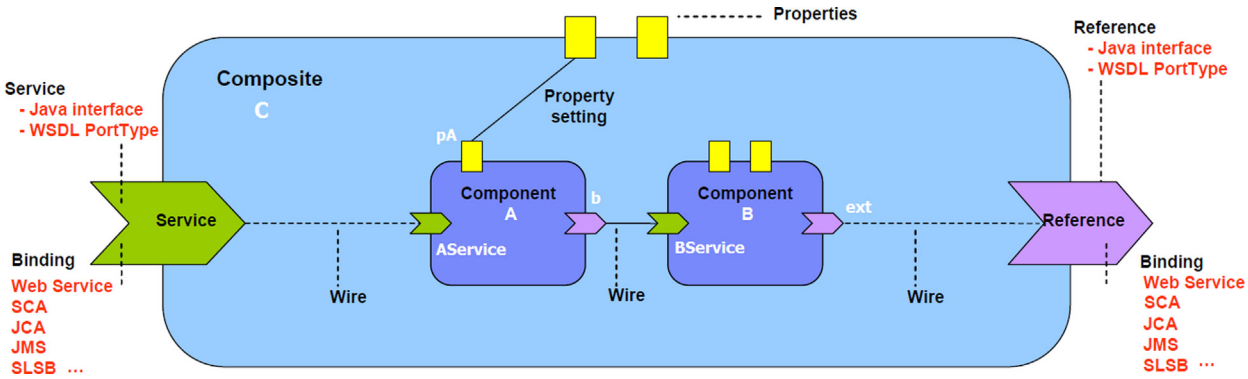


Fig. 1. An SCA composite (adapted from the SCA Assembly Model V1.00 spec.).

For behavioral specification and functional analysis purposes, we in particular adopt the *SCA-ASM component implementation type* [23]. SCA-ASM is based on a suitable subset of the SCA component model and complements it with the model of computation of the formal method *Abstract State Machine* (ASM)[2] [26] to model notions of service behavior, interactions, orchestration, and compensation in an abstract, formal, and executable way. An open framework, the ASM toolset ASMETA (ASM mETA-modeling) [27,28], based on the Eclipse/EMF platform and integrated with the SCA runtime Tuscany, is also available for editing, simulating, validating, and potentially model checking SCA-ASM models [29,30].

3.1.1. Adaptation mechanism

Adaptation actions at technology or business level are easily supported by extending/modifying the SCA assembly of the application. Mechanisms for enacting and managing self-adaptation of SCA applications are being developed for SCA run-time platforms. The work [31] is an example of such an effort showing how such mechanisms are being implemented in the SCA runtime platform FraSCAti.

Our SCA-based framework achieves adaptations through the insertion or replacement of components or sets of components or services. Basically, an SCA assembly (i.e., the architecture of the considered application) can be adapted through the following atomic *adaptation actions*:

- adding/removing components, component services, references, properties, and wires (component interactions);
- changing a component implementation (but keeping its shape);
- changing component properties values;
- changing SCA domains[3] (components re-deployment);
- changing the component interaction style in synchronous/asynchronous, stateful or not, unidirectional or bidirectional.

The changes in the interaction style are reflected at SCA level by changing the shape, at interface level, of the components involved in the interaction and the wire type (communication binding) used to interconnect the components. See [7] for more details. Moreover, with respect to changes in the system behavior (that is formally specified in terms of ASMs for functional analysis purposes), an adaptation action may imply also changes in the services interaction(s) and orchestration process. These are reflected also at ASM level, as refinement of the ASM transition rules specifying the components' services behavior and their orchestration.

Adaptations actions can be combined into an *adaptation plan*, which is a set of actions modifying the static and dynamic parts of an application architecture to address a certain requirement. Adaptation plans may differ for adaptation cost and/or for the system quality achieved after their application.

3.2. Adaptation Space Exploration

In this section, we describe an optimization process that suggests how to adapt the architecture of a service oriented application by making the right tradeoff between functional and non-functional requirements. Specifically, cost and multiple quality attributes (such as response time and availability) are considered.

The workflow of the proposed process is shown in Fig. 3. It starts (*Candidates Selection*) with an initial set of SCA-ASM assemblies (the initial *candidates* or *population* of the search) fulfilling the existing/new functional requirements and fully QoS annotated. It then proceeds iteratively. At each iteration step, new candidates are generated from the current population

[2] ASMs are an extension of FSMs: states are arbitrary complex data (multi-sorted first-order structures) and the transition relation is specified by rules describing how functions change from one state to the next.

[3] An SCA domain consists of the definitions of composites, components, their implementations, and the nodes on which they run. Components deployed into a domain can directly wire to other components within the same domain.

1. Model Parameters and Variables

n	number of system components
m	COTS instances available for each component
c_i	unitary development cost
t_i	develpment time
τ_i	average time required to perform a test case
s_i	average number of invocations
μ_{ij}	probability of failure on demand
ρ_i	probability that the in-house instance is failure free
d_{ij}	delivery time
y_i	the component i is in-house developed
x_{ij}	COTS Instance j selected for the component i
N_i^{tot}	number of tests performed on the in-house developed instance of the component i

2. Cost Objective Function: $COF = \sum_{i=1}^{n} \left(c_i(t_i + \tau_i N_i^{tot})y_i + \sum_{j=1}^{m} c_{ij} x_{ij}\right)$

3. Reliability Constraint: $RelSys = \prod_{i=1}^{n} e^{-((1-\rho_i)s_i y_i + \sum_{j=1}^{m} \mu_{ij} s_i x_{ij})} \geq R$

4. Delivery Time Constraint: $\max_{i=1,\ldots,n} (T_i) \geq T$

5. The delivery time of the component i: $T_i = y_i(t_i + \tau_i N_i^{tot}) + \sum_{j=1}^{m} d_{ij} x_{ij}$

Fig. 2. Cost Model and Delivery Time/Reliability constraint used in [33].

(*New Candidates Generation*), then evaluated (*Functional and Quality Analysis*) and the promising ones selected (*Candidates Selection*), until the stop criteria are satisfied[4] [32].

There are two methods to generate new candidates based on the current population, which are executed in parallel:

- (i) *Metaheuristic Search*;
- (ii) *Application of Existing Design Solutions*

Both these two methods contribute new candidates to the next iteration. The first method uses the classic metaheuristic search techniques to explore the adaptation space by applying user adaptation plans, service selection and service re-deployment. The size of the population depends on the specific search technique. The second method uses the specific knowledge of architectural design patterns and tactics to improve extra-functional aspects in current candidates, if applicable. Essentially, new SCA-ASM assemblies are created from the current ones by making the necessary adaptation actions required by the application of the specific design pattern or tactic. This may imply also the introduction of new concrete components available from a component repository.[5]

The functional and quality analysis of the new population of candidates are performed together with an assessment of the adaptation costs. Such analysis can also be used during the execution of the subprocesses to optimize the search space exploration. For example, as done in the example shown below, the search metaheuristic process is guided by reliability analysis. For each candidate, the metaheuristic algorithm performs the quality analysis by using the reliability model and cost model used in [33]. As shown in Fig. 2, the cost model – representing the objective function – and the reliability model – representing a model constraint – allows to predict the system reliability and cost as a function of the reliability (cost) of single components.

In case of self-adaptation, the promising SCA-ASM assemblies are automatically selected as solutions according to pre-defined selection criteria (e.g., cost minimization). In case of evolution, the solution can be more accurately selected also considering a possible feed-back from the user [9].

Example of Adaptation Process Fig. 4 shows a concrete example of instantiation of our adaptation process, where an UML Sequence Diagram (SD) describes the process behavior in terms of interactions that take place among modules executing the main Activities shown in Fig. 3 (i.e., *Candidate Selection*, *Metaheuristic Search*, *Application of Existing design solutions* and *Functional and Quality Analysis*). The application under consideration is composed by the components C_1, C_2 and C_3. The

[4] We currently use a predefined number of iterations to determine the end of the search. More sophisticated stop criteria could use convergence detection and stop when the global optimum is probably reached.

[5] We assume that a number of concrete components are available in some repository, which is accessible to us and we need to make a selection out of them.

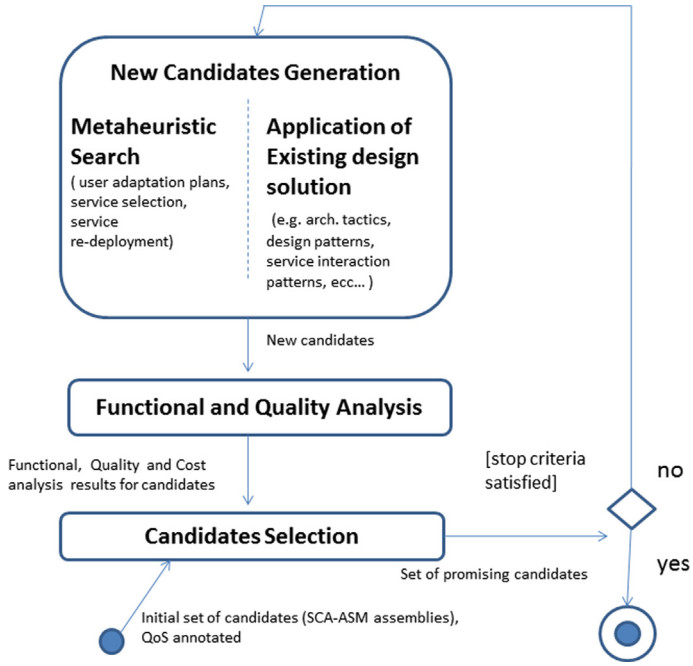


Fig. 3. Adaptation Space Exploration.

adaptation process tries to adapt the application minimizing the adaptation costs and assuring a level of reliability greater than 0.99. The initial candidate of the process – provided to the module *Candidates Selection* for starting the adaptation space exploration, and generated by replacing each application component with one of its three available instances – is the vector $[C_{11}, C_{21}, C_{31}]$ (see Fig. 4(a)), where C_{ij} denotes the j-th available instance for the component C_i. The vector comes with two parameters: the resulting system reliability and the cost of the solution. □

In order to support such proposed optimization process, we have also been working on developing a suitable software-based "reasoner module" that can be embedded in any general-purpose adaptation framework. This software module (see Fig. 5) may be activated after receiving (as input) either adaptation requests from the user or alerts from monitors (or probes). By executing the process described above, this reasoner produces as output Pareto-efficient alternatives. Each point of a Pareto curve is a chain of software adaptation actions that, if executed, lead to the new system architecture, i.e., the adapted SCA-ASM assembly. These changes applied at SCA model level must be related then (through the use of effectors) to the underlying mechanisms and runtime infrastructure. In the case of SCA, mechanisms like introspection and reconfiguration, for managing and enacting self-adaptation of SCA assemblies [31] are applied.

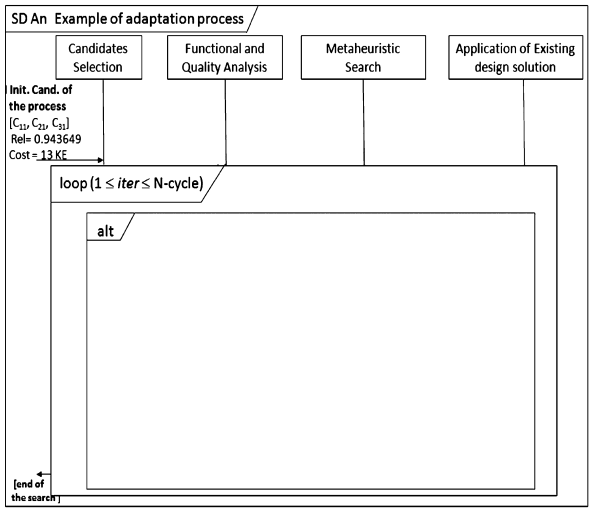
Below, a more detailed description of the main phases follows.

New candidates generation: As first generation method, we rely on the use of metaheuristic search techniques. Their effectiveness and efficiency has been already demonstrated [6] for supporting the service selection activity at run-time. As remarked in [6], the global optimization, typically used by the approaches supporting such an activity driven by system quality, is definitely useful for small composition, but a significant performance penalty incurs for large-scale optimization problems, especially for runtime optimization. Several metaheuristic techniques [34] with different characteristics could be adopted depending on the nature of the problem. For example, considering the system reliability, a possible heuristic is: *"the reliability of the whole system increases when the reliability of the most used components increases"*. As remarked in [35], there exist design options for which we have no prior knowledge on how they affect the extra-functional property of a particular system. To this extent, undirected operations could be performed (e.g., random choices or exhaustive evaluation of all neighboring candidates).

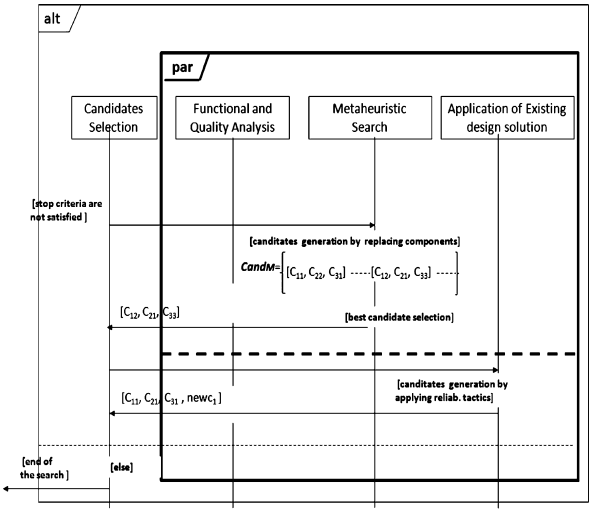
In our optimization approach, we adopt the *Tabu Search* heuristic technique described below, in Section 3.3. The result of the first generation method is therefore a set of Pareto-optimal candidate solutions, probably superior to the initial candidates.

For comparison purposes only (see the experimentation in Section 6), we also consider two other methods that are described in Section 3.3.2.

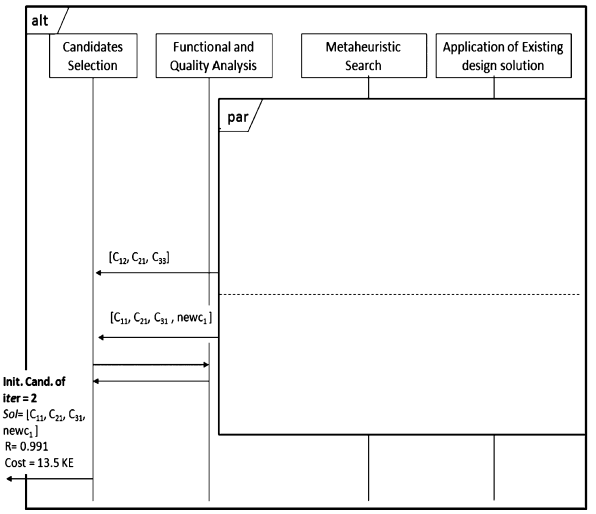
Example of Adaptation Process At the first iteration step of the adaptation process, the module *Metaheuristic Search* (see Fig. 4(c)) generates the set of new candidates $Cand_M$. The module *Metaheuristic Search* selects the best candidate as the one improving the reliability and minimizing the adaptation costs. It becomes the basis for next candidates generation. The process terminates either if no better candidates can be found or the reliability threshold is reached. In our case, the metaheuristic returns the solution $[C_{12}, C_{21}, C_{33}]$ with reliability equals to 0.990644 and cost equals to 14.5 (in KiloEuros, KE). □



(a) The starting point of the process



(b) The parallel execution of two methods at the first process's iteraction



(c) The second iteration of the process

Fig. 4. An example of adaptation process.

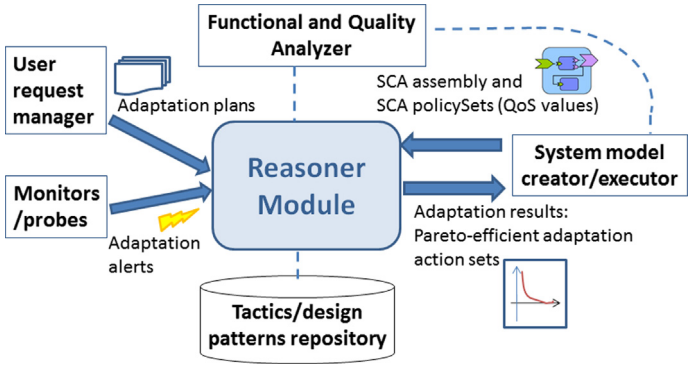


Fig. 5. Reasoner module.

The second method generates new candidates by applying recurring software design solutions, such as architectural design patterns and tactics. Architectural design patterns are templates for concrete software architectures. They are adopted to embody functional requirements and, in particular, to enable self-adaptability by introducing sensors/effectors components (e.g., Microkernel pattern, Reflection pattern, Interception pattern) [10]. To build new design solutions embodying extra-functional requirements, we adopt architectural tactics [36], which are reusable architectural building blocks that provide a generic solution to address issues pertaining to quality attributes.

Example of Adaptation Process At the first iteration step of the adaptation process, concurrently to the metaheuristic search, the module *Application of existing design solution* (see Fig. 4(c)) applies tactics for reliability improvement leading to a change of the components shape (i.e., in the required/provided interfaces) and also of their behavior. Besides, it is required the introduction of a new component $newc_1$ for the fault detection tactics. □

Functional and Quality Analysis: This phase aims at guaranteeing the functional correctness of the resulting candidates and that changes claimed by the adaptation actions do not compromise the satisfaction of existing non-functional requirements. To this purpose, we use a set of external analysis tools that can be invoked for the functional and non-functional analysis, respectively. The *functional analysis* is assumed (but not limited to) that is executed with the help of the ASM analysis toolset ASMETA [27,37]. Preliminary analysis of the functional requirements satisfiability of an SCA-ASM assembly would be performed by easier validation techniques as simulation, [38] scenario-based simulation [39,40], and model-based testing [41]. Later, heavier formal verification techniques (as model checking) [42] can be exploited when more complex functional properties [43] must be proved to guarantee behavioral system correctness.

For the *non-functional analysis*, external tools for performance and reliability analysis like `qnetworks` [44] and `LQN-solver` [45] are exploited. Examples of adaptation costs prediction techniques can be found in [2]. In general, different approaches/strategies can be used depending on several factors due mainly to the use of our process for evolution (at re-design time) or self-adaptation (at run time). If permanent changes, for example, are requested or a safe-critical service has to be adapted, precise and often expensive analysis must be performed (e.g., see [5] for performance analysis). As opposite, if runtime changes are claimed and these require, for example, only the adaptation of parameters without using more sophisticated analysis, faster approaches must be adopted allowing a prompt run-time adaptation (see, e.g., techniques for estimation of quality at runtime, such as [46]). Moreover, as we remarked above, this analysis could be also performed during the execution of the two subprocesses executed in parallel. Similar considerations also apply for functional analysis.

Example of Adaptation Process As shown in Fig. 4(c), the *Candidates Selection* module uses the module *Functional and Quality Analysis* for the quality analysis of the candidates generated by the module *Application of Existing design solutions*. As already stated, the module *Metaheuristic Search* performs itself the quality analysis.

The best candidate is selected (by the module *Candidates Selection*) among the resulting candidates of the two subprocesses as the one improving the system reliability and minimizing the adaptation costs. It becomes the basis for the second candidates generation. As shown in Fig. 4(d), the initial candidate of the second iteration of the exploration process is the solution $Sol = [C_{11}, C_{21}, C_{31}, newc_1]$ provided by the tactics applications with reliability equals to 0.991 and cost equals to 13.5 KE. The process terminates till stop criteria are satisfied (i.e., the number of iterations determining the end of the search is greater than Ncycles). □

3.3. Multi-objective optimization and Pareto solutions

As already stated, we deal with multi-attribute problems and the proposed optimization process exploits the multi-objective optimization [47], where the objectives represent different quality attributes. The aim of these techniques is to devise a set of solutions, called Pareto optimal solutions or Pareto front, each of which assures a trade-off between the conflicting qualities. In other words, while moving from one Pareto solution to another, there is a certain amount of sacrifice in one objective(s) to achieve a certain amount of gain in the other(s).

The *adaptation space AS*, that is the search space of our optimization process (i.e., the set of all possible candidate solutions), is the Cartesian product of the option sets of all system-specific adaptation actions.
In our context, we can state that a candidate SCA-ASM assembly is Pareto-optimal, if it is superior to any candidates evaluated so far in at least one quality criterion.

More formally: Let s be a candidate solution, let $C \subseteq AS$ be a set of candidate solutions evaluated so far, and let q be a quality criterion with a domain D_q, and an order $\leqslant_q$ on D_q so that $s_1 \leqslant_q s_2$ means that s_1 is better than or equal to s_2 with respect to quality criterion q. Then, candidate solution s is Pareto-optimal with respect to a set of evaluated candidate solutions C, iff

$$\forall s' \in C \exists q\colon\ f_q(s) \leqslant_q f_q(s')$$

If a candidate solution is not Pareto-optimal, then it is Pareto-dominated by at least one other candidate solution in C that is better or equal in all quality criteria. Analogously, a candidate is globally Pareto-optimal, if it is Pareto-optimal with respect to the set of all possible candidates AS.

In the remainder of this section we review the main methods used in the process to find these Pareto optimal solutions.

```
1.   s ⟵ GenerateInitialSolution()
2.   TabuList ⟵ ∅
     // s′ memorizes the best solution of
     // the tabu search
3.   s′ ⟵ s
4.   while termination conditions not met do
5.     NeighboursOkSet ⟵ ExploreNeighbourhood(s)
6.     s ⟵ ChooseBestof(NeighboursOkSet)
7.     Update(TabuList)
8.     if (f(s′) > f(s)) then
9.       s′ ⟵ s
       end if
     end while
```

Fig. 6. Algorithm: Tabu Search.

3.3.1. Tabu Search

The Tabu Search (TS) [34] is among the most cited and used metaheuristics for solving optimization models. It enhances the performance of a local search method by using memory structures describing the visited solutions. Once a solution is visited, it is marked as "taboo" so that the TS does not visit that possibility repeatedly. TS explicitly uses the history of the search, both to escape from local minima and to implement an explorative strategy. For a given problem, the solution produced by TS algorithm mainly depends on the combination of its parameters (e.g., the neighborhood size, the number of the iterations and tabu tenure).

The pseudo-code of the simple TS algorithm is shown in Fig. 6. A description of its main steps follows.

Begin with a starting current solution The initial candidate s, representing an SCA-ASM assembly and fulfilling the existing/new functional and non-functional requirements is generated.

Create new candidates The tabu search is based on a *short term* memory, which is implemented as a *tabu list*. This latter keeps track of the most recently visited solutions and forbids moves toward them.

At each iteration step, the neighborhood of the current solution[6] is restricted to the solutions that do not belong to the tabu list (i.e., definition of *NeighborsOkSet* in Fig. 6). Such a set of new candidates is obtained making changes to the current solution (these changes are also called *moves*) by applying user adaptation plans, service selection and service re-deployment.

Choose the best candidate The best candidate is then selected as the one minimizing the objective function (under possible constraints). This step is performed through the function *ChooseBestof*(*NeighboursOkSet*) in Fig. 6. The candidate becomes the basis for next candidates generation and the current best solution of all tabu search interactions. Additionally, such a solution is added to the tabu list and one of the solutions that were already in the tabu list is removed (usually in a FIFO order). The length of the tabu list is given as value of input to the tabu search.

Stopping criterion The process proceeds iteratively till stop criteria are satisfied by returning the best solution of all interactions. The algorithm can stop if the predefined number of iterations has elapsed in total. More sophisticated stop criteria could use convergence detection and stop when the global optimum is probably reached.

This simple TS could be specialized and enhanced depending on the problem, varying, for example, the tabu list length or leveraging on long-term memory [34].

3.3.2. Other methods

For comparison purposes, in our experimentation we considered two other methods to generate alternative adaptation solutions: the *lexicographic method* [48], and a *random solution generator*.

We have implemented the *lexicographic method* as follows. First we have solved the optimization model minimizing the adaptation cost under reliability and delivery time constraint (using the model in [33]), then we have formulated the optimization model that minimizes the probability of failure under the cost constraint expressed as $f_1(x) \leqslant f_1(x*) + \epsilon$, where ϵ is a positive tolerance (real number). Finally, we have found the set of Pareto optimal solutions by varying ϵ applying the ϵ-constraint approach [48]) The process continues till the stop criteria are satisfied.[7] For the experimentation we have used the LINGO tool [49], which is a non-linear model solver, to produce the results.

The choices of the *random solution generator* were, of course, random. For the component C_i, a COTS component is chosen by picking it uniformly at random from the set of COTS instances available for C_i.

4. Running case study

This section presents a sample application from the Stock Trading System (STS) originally presented in [36]. It is used throughout the article to exemplify the proposed adaptation process. Fig. 7 shows the initial SCA assembly of the STS. Briefly, an STS user, through the `OrderWebComponent` interacting with the `OrderDeliveryComponent`, can check the

[6] "A neighborhood structure is a function N: $S \to 2^S$ that assigns to every $s \in S$ a set of neighbors $N(s) \subseteq S$. $N(s)$ is called the neighborhood of s." [34].

[7] We have used the satisfaction of reliability constraint to determine the end of the search and a predefined number of interactions.

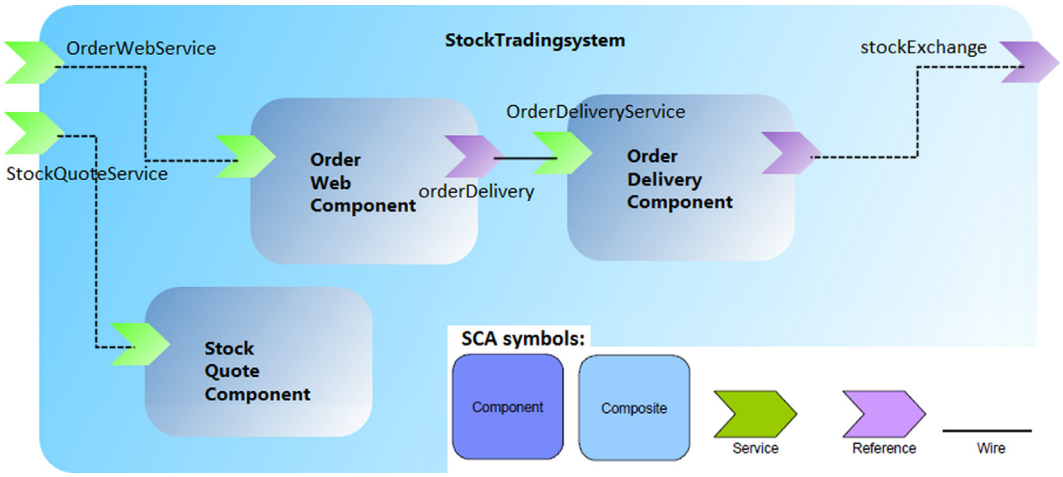


Fig. 7. Stock Trading System.

Table 1
Parameters of instances available for existing components.

	Instance alternatives	Cost c_{ij}	Average delivery time d_{ij}	Average no. of invocations s_i	Prob. of fail. on demand μ_{ij}
C_1	C_{11}	1	4	180	0.0002
	C_{12}	2.5	4	180	0.0002
	C_{13}	2	4	180	0.0004
C_2	C_{21}	2	4	20	0.0002
	C_{22}	3	4	20	0.0002
	C_{23}	6	15	20	0.0004
C_3	C_{31}	10	4	60	0.0002
	C_{32}	14	10	60	0.0004
	C_{33}	10	10	60	0.0004

current price of stocks, placing buy or sell orders and reviewing traded stock volume. Moreover, he/she can know stock quote information through the `StockQuoteComponent`. STS interacts also with the external Stock Exchange system, which we do not model.

Below we report the experimental data set used in this case study.

4.1. Model parameters of the STS case study

The estimation of functional/extra-functional parameters is a well-know problem. It has been discussed, for example in [50], where the state of existing approaches is identified. Possible solutions are proposed, mainly based on the usage of historical data (if available) or derived from similar situations.

The model parameters used here are partially derived from the WEB-based data retrieval system adopted as case study in [33]. In such a system clients are equipped with the local processing capability and local database. A WEB connection is provided to execute data search and data updates in a distributed system, through the network. We have assumed, though not shown in Fig. 7, that our STS application also instantiates an agent on the server side of the WEB-based data retrieval system.

As done in [33], the estimation of the parameters entering our system has been partly based on the monitoring of an existing data retrieval system at University of L'Aquila and partly extracted from software artifacts of the same system. However, incomplete documentation forced us to adopt extrapolation techniques for providing certain values. For example, the number of invocations has been obtained by analyzing partial scenario descriptions and validating the analysis results with monitored average number of interactions.

We assume that several instances (i.e., implementations) of an existing component may be available on the market, all *equivalent* from the functional viewpoint. Basically, the instances differ each other for costs and non-functional properties such as reliability and response time. Table 1 shows the initial values that we have considered for the parameters of SCA components available on the market. Similarly, Table 2 shows the initial parameters that we have considered for in-house SCA components. We have associated the IDs to the components as follows: C_1 to *Order Web Component*, C_2 to *Stock Quote Component* and C_3 to *Order Delivery Component*.

The second column of Table 1 lists, for each component, the set of instance alternatives available. For each alternative: the adaptation cost c_{ij} (in KiloEuros, KE) is given in the third column, the average delivery time d_{ij} (in time unit) is given

Table 2
Parameters for in-house developed instances.

	Development time t_{i0}	Testing time τ_{i0}	Unitary development cost $\bar{c}_{i0}$	Average no. of invocations s_i	Testability π_{i0}
C_1	1	0.05	1	180	0.002
C_2	3	0.05	1	20	0.002
C_3	5	0.05	1	60	0.002

in the fourth column, the average number of invocations of the component in the system s_i is given in the fifth column, finally the probability of failure on demand μ_{ij} is given in the sixth column.

For each component in Table 2: the estimated development time t_{i0} (in time unit) is given in the second column and the average time required to perform a test case τ_{i0} (in time unit) is given in the third column, the unitary development cost $\bar{c}_{i0}$ (in KE per day) is given in the fourth column, the average number of invocations s_i is given in the fifth column, and finally the component testability π_{i0} is given in the last column. The definition of testability π_i that we adopt is the one given in [51], that is the probability that a single execution of a software fails on a test case chosen from a certain input distribution. The input distribution represents the operational profile that we assume for the component, as obtained from the operational profile of the whole application [52].

Note that we have defined the parameters of components as average values of the values of their provided services. The parameters could be refined with respect to the services without essentially changing the overall model structure.

STS system configurations. We here discuss the parameters of three configurations of the STS system used later in Section 6.2.

The first configuration has the threshold on the delivery time and reliability to $T = 7$ and $R = 0.5$, respectively. In addition, the costs of C_{11} and C_{21} is increased to 5 units (i.e., C_{11} and $C_{21} = 5$). The reliability threshold (that may be unrealistic) has been set to this value to show the behavior of three approaches in the case of a not complex search space.

The second configuration has the threshold on the delivery time and reliability to $T = 15$ and $R = 0.8$, respectively. In addition, the costs of C_{11}, C_{21} and C_{22} are increased to 5 units (i.e., $C_{11} = C_{21} = C_{22} = 5$), and the probability of failure on demand of SCA component implementations available for C_1 is increased to 0.003 (i.e., $\forall j\ \mu_{1j} = 0.003$).

Finally, the third configuration has the threshold on the delivery time and reliability to $T = 15$ and $R = 0.8$, respectively. In addition, the costs of C_{11}, C_{21} and C_{22} are increased to 5 units (i.e., $C_{11} = C_{21} = C_{22} = 5$), and the probability of failure on demand of SCA component implementations available for all components is increased to 0.003 (i.e., $\forall i, j\ \mu_{ij} = 0.003$).

The experiments were run on a Windows workstation equipped with an Intel Centrino Processor 1.3 GHz CPU and a 512 MB RAM. The tabu search algorithm was compiled using lcc-win32 3.3. We imposed a number of interactions of 50, and the tabu list length limit of 45 to each experiment. Finally, the optimization model for the lexicographic method was solved using LINGO 11.0.

5. Adaptation space exploration for service-oriented components replacement

In this section, we exemplify the proposed adaptation methodology by instantiating the tabu search for the specific problem of service-oriented components replacement. We illustrate the adaptation process through the STS case study where most adaptation actions concern the replacement of SCA components.

5.1. Downscaling the Tabu Search for SCA components replacement

This tabu search application is designed for replacing the STS components by buying or building components on the base of cost and non-functional factors (i.e., reliability and delivery time). The TS also provides the best amount of testing to be performed on each in-house developed component to fulfill the constraints while minimizing the adaptation costs. The TS solves the non-linear cost/quality optimization model [33] based on decision variables indicating the set of architectural components to buy and to build in order to minimize the software cost under the delivery time and reliability constraint (i.e., the system reliability and delivery time are required within a threshold R and T, respectively). Such a model belongs to the class of mixed integer non-linear programming models. It can suffer of large elapsed time when its size increases (e.g., it grows exponentially in the number of components). We have implemented in C and optimized for fast execution the TS algorithm. The entire set of experiments, which we have performed, took practically no noticeable time (order of seconds) on standard computing equipment.

In the sequel we describe how the steps of the general algorithm described in Section 3.3.1 are instantiated.

Multi-objective function As shown in Fig. 8, the function objective $f(s)$ is the weighted sum of the adaptation cost and the system probability of failure obtained by considering the candidate s. These objectives are defined by using the cost model and the reliability model used in [33] (see Fig. 2). In particular, at each iteration step, the neighborhood of the current solution is explored (i.e., for each neighbor s, $f(s)$ is estimated). The best candidate is then selected as the one minimizing the objective function (under possible constraints). This step is performed through the function ChooseBestof of Fig. 6. The tabu search is executed for different set of s values. The solutions of each tabu search's execution are analyzed,

$$\textbf{Objective Function:}\quad f(s) = \alpha_1 f_1(s) + \alpha_2 \log f_2(s)$$
$$f_1(s) = \text{COF}$$
$$f_2(s) = 1 - \text{Relsys}$$
$$\sum_{i=1}^{2} \alpha_i = 1$$
$$\alpha_i > 0, i = 1 \ldots 2$$

Fig. 8. Objective function as weighted sum of the cost model and reliability model used in [33].

and the set of Pareto-optimal solutions is defined. Note that, to sum such objectives we apply the logarithm function to the system probability of failure on demand. In fact, since the probability of failure on demand is a number that falls within the range of [0, 1] its logarithm is a negative number.

Begin with a starting current solution The initial candidate s, which fulfills the reliability and delivery time constraints is the vector $[C_{ij}]$, made of three elements, where an element C_{ij} denotes either a COTS[8] or an in-house instance. For its first application, C_{i0} means an in-house developed component and the symbol name of the instance is paired with the number of test to perform on the instance.

C_1 indicates the `OrderWebComponent`, C_2 the `StockQuoteComponent` and C_3 the `OrderDeliveryComponent`. The resulting system reliability, the cost and delivery time of the solution are predicted using the reliability, cost and delivery time model used in [33] (see Fig. 2). The candidate s is generated by using the *TS Initial Solution Generation* method described in Section 5.1.1.

Create new candidates The set of new candidates is generated by replacing, one at a time, an existing component with either one available on the market or an in-house instance. The amount of testing of the in-house developed instances is found by using the *Testing Generation* algorithm described in Section 5.1.2.

Choose the best candidates The best candidate is selected as the one minimizing the objective function under reliability and delivery time constraints. Additionally, the pair (i, j) is stored into the tabu list, where i is the index of component changed and j represents the new solution chosen for the component i, and the oldest components in the tabu list are removed as the list becomes full following a FIFO order.

Stopping criterion The TS stops if the predefined number of iterations has elapsed in total. At each iteration, upon examining the neighborhood, if no feasible solution is found, then the initial solution of the next interaction is generated using the *TS Initial Solution Generation* method.

5.1.1. Initial Solution Generation algorithm

Depending on several factors (e.g., search technique used) different strategies could be adopted [34] for finding a set of feasible solutions, such as an algorithm combining heuristics, local search, user adaptation plans, service selection and re-deployment actions.

For this step, we draw inspiration from the solution construction phase of the Greedy Randomized Adaptive Search Procedure (GRASP) [34]. We generate a list of feasible solutions that fulfill the quality constraints (i.e., the system reliability and delivery time are assured within the required thresholds). A solution is the vector $[C_{ij}]$ where an element C_{ij} denotes either a COTS instance or an in-house instance. First the COTS component C_{1j} is chosen by picking it uniformly at random from the set of COTS instances available for C_1, then the set of solutions is generated by replacing, one at a time, another existing component with one available on the market instance.

Such a search is optimized by the reliability heuristic described above. If no solutions are returned, the process is repeated by choosing another component $C_{1\bar{j}}$ using the reliability heuristic, and a solution is randomly generated by considering also the in-house instances.

5.1.2. Testing Generation algorithm

The *Testing Generation* (TG) algorithm estimates the amount of testing of the n_{house} in-house developed instances of the candidate s. In the following we discuss the main steps and features of TG, which implements another tabu search.

Begin with a starting current solution The initial candidate t, that fulfills the quality constraints is the vector $[t_h]$, made of n_{house} elements, where an element t_h denotes the maximum amount of unit test $maxt_h$ that could be performed on the in-house component h. t_h is estimated as a function of the threshold T required for the system delivery time.

Create new candidates At each iteration step, the neighborhood of the current solution t is generated by varying the testing of an in-house instance h on the range $[0, maxt_h]$.

Choose the best candidate The best candidate is selected as the one minimizing the objective function $f(s)$ under reliability and delivery time constraints. The candidate becomes the basis for next candidates generation and possibly the current best solution of all TG interactions. Additionally, the pair (h, t_h) is added to the tabu list, where h is the index of the changed in-house element (with respect to the initial candidate t of the current TG interaction) and t_h represents the

[8] Throughout this article, we use improperly the term COTS to denote in a concise way an SCA service-oriented component already available on the market.

amount of unit test for the in-house h. The oldest solutions that were already in the tabu list are removed whether the list is full using a FIFO discipline. The length of the tabu list is given as value of input to the tabu search.

Stopping criterion The TG stops if the predefined number of iterations has elapsed by returning the best solution of all interactions. At each interaction, upon examining the neighborhood, if no feasible solution is found, then the initial solution for the next iteration is generated as follows.

A starting solution $t = [t_h]$ is generated with an algorithm similar to the Initial solution algorithm described above. First t_1 is chosen by picking it uniformly on the range $[0, maxt_1]$, then the other testing amounts are generated by varying the one of an in-house instance h ($h \neq 1$) on the range $[0, maxt_h]$. Note that such an algorithm could be also used for generating the initial candidate for the first interaction of TG.

6. Adaptation space exploration of the STS

In this section, we illustrate the experimental results of a possible execution of the proposed adaptation process for the STS case study, while in the next section we show the results of an extensive numerical evaluation conducted to analyze the performance of the overall process. Specifically, first we show the application of the TS described in the previous section to the STS. We also confirm the importance of having such an automated support by showing that our heuristic search produces results that exceed the one produced by the other two considered methods, i.e., the lexicographic method and the random solution generator. Then, we combine the use of the metaheuristic search with some tactics and architectural adaptation patterns, re-iterating the process.

6.1. Application of the Tabu Search to the STS

We have designed a TS for replacing the STS components by buying or building components basing on cost and non-functional factors (i.e., reliability and delivery time). The TS also provides the best amount of testing to be performed on each in-house developed component to fulfill the constraints while minimizing the adaptation costs. The TS solves an optimization model as the one proposed in [33]. The search space (i.e., the set of all possible candidate solutions) is the Cartesian product of the COTS and in-house instances sets of the STS components. The instantiation of TS to the case study has been described in Section 5.1. Each candidate is expressed as a vector $[C_{ij}]$. In the STS case study, C_1 indicates the `OrderWebComponent`, C_2 the `StockQuoteComponent` and C_3 the `OrderDeliveryComponent`. Detail on the TS implementation can be found in [53].

6.2. A comparison among the tabu search, the lexicographic method, and the random solution generator

We have applied the approaches on three different configurations of the STS system characterized by the parameters reported in Tables 1 and 2. In order to keep our model as simple as possible, in all configurations we assume that only one in-house instance for each component can be developed, but several in-house components can exist. The number of COTS instances does not change across configurations, but each configuration is based on a different set of component parameters. The configurations differ also for the values of reliability R and delivery time T bounds.

The configuration parameters have been set for showing the behavior of three approaches while increasing the search space complexity. The configurations differ for the probability to find a Pareto solution: the first configuration, characterized by a lower threshold R, has a higher probability with respect to the other ones, characterized by a higher threshold R and a set of selected components more complex to be analyzed.

Fig. 9 shows the approximate Pareto curves obtained from solving the optimization problem of the first system configuration using the three approaches. In the figure we have put a cross on the unfeasible solutions. In this configuration a maximum threshold $T = 7$ has given on the delivery time of the whole system, and a minimum threshold $R = 0.5$ is given on the reliability of the whole system. This reliability threshold (that may be unrealistic) has been set to show the behavior of three approaches in the case of a not complex search space.

Each Pareto solution represents a configuration of components that minimizes both the system adaptation cost and its probability of failure on demand. For example, the tabu search results claim that if the probability of failure is equal to 0.405479 (represented on the x-axis), then the minimum cost to adapt the system is 9 KE. The tabu search results also show that the optimal solution cost increases (up to 17.5 KE) while decreasing the system probability of failure (down to 0.050671).

By looking at the details of the solution, for example, we observe that for the tabu search point (0.405479, 9 KE) the solution point is: $[(C_{10}, 0), (C_{20}, 0), (C_{30}, 0)]$. This means that, in order to achieve the optimal cost of adaptation all components have to be built in-house and without suggesting an amount of testing. We recall that the amount of testing is suggested only when it is necessary to make an instance more reliable in order to achieve the desired level of reliability R.

Table 3 reports further details on the results shown in Fig. 9. The table is organized as follows: the first, second and third columns represent the lexicographic method, the tabu search and the random generator, respectively. In each entry (row, column) we represent the choice of components (i.e., a Pareto solution). The choice is represented as a vector, where each element can be either a COTS instance or an in-house instance. For its first application, C_{i0} means an in-house developed component and the symbol name of the instance is paired with the number of successful (i.e., failure-free) test to perform

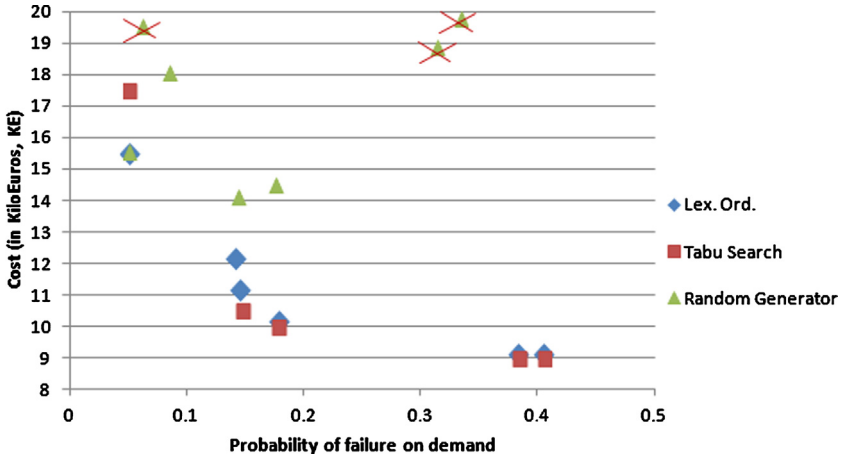


Fig. 9. Comparison of the three approaches with respect to the first configuration.

Table 3
Results from Lexicographic, Tabu Search, and Random Generator for the first configuration.

Lexicographic	Tabu Search	Random Generator
$[(C_{10}, 1), (C_{20}, 1), (C_{30}, 1)]$ $FRsys = 0.404862$ $Cost = 9.15$ KE	$[(C_{10}, 0), (C_{20}, 0), (C_{30}, 0)]$ $FRsys = 0.405479$ $Cost = 9$ KE $Time = 13.06$ sec, $\alpha = 1$	$[C_{11}, C_{22}, (C_{30}, 22)]$ $FRsys = 0.143447$ $Cost = 14.102$ KE
$[(C_{10}, 3), C_{22}, (C_{30}, 0)]$ $FRsys = 0.382359$ $Cost = 9.15$ KE	$[(C_{10}, 0), C_{22}, (C_{30}, 0)]$ $FRsys = 0.383687$ $Cost = 9$ KE $Time = 14.482$ sec, $\alpha = 0.6$	$[C_{12}, C_{21}, C_{32}]$ $FRsys = 0.0506711$ $Cost = 15.5$ KE
$[C_{13}, C_{22}, (C_{30}, 3)]$ $FRsys = 0.177398$ $Cost = 10.15$ KE, $\epsilon = 0.999$	$[C_{13}, C_{22}, (C_{30}, 0)]$ $FRsys = 0.177988$ $Cost = 10$ KE $Time = 11.938$ sec, $\alpha = 0.4$	$[C_{12}, C_{22}, C_{32}]$ $FRsys = 0.0619950$ $Cost = 19.5$ KE
$[C_{12}, C_{22}, (C_{30}, 13)]$ $FRsys = 0.145230$ $Cost = 11.15$ KE, $\epsilon = 2$	$[C_{12}, C_{22}, (C_{30}, 0)]$ $FRsys = 0.147856$ $Cost = 10.5$ KE $Time = 11.998$ sec, $\alpha = 0.2$	$[(C_{10}, 25), (C_{20}, 10), C_{32}]$ $FRsys = 0.333468$ $Cost = 19.753$ KE
$[C_{12}, C_{22}, (C_{30}, 33)]$ $FRsys = 0.141306$ $Cost = 12.15$ KE, $\epsilon = 3$	$[C_{12}, C_{21}, C_{31}]$ $FRsys = 0.050671$ $Cost = 17.5$ KE $Time = 9.855$ sec, $\alpha = 0$	$[C_{11}, (C_{20}, 1)C_{31}]$ $FRsys = 0.0841660$ $Cost = 18.050$ KE
$[C_{12}, C_{22}, C_{31}]$ $FRsys = 0.0506711$ $Cost = 15.5$ KE, $\epsilon = 8$		$[(C_{10}, 17)C_{22}, C_{32}]$ $FRsys = 0.313382$ $Cost = 18.851$ KE
		$[C_{11}, (C_{20}, 19), (C_{30}, 10)]$ $FRsys = 0.174805$ $Cost = 14.452$ KE

on the instance. Beside the vector of instance components, the resulting system probability of failure (i.e., *FRsys*) and the cost (i.e., *Cost*) of the solution are reported in each entry. Furthermore, specific parameters (i.e., α_i, ϵ, and the tabu search execution time *Time*) of the approaches are also reported.

The results highlight, in general, that the solutions of the lexicographic method and the tabu search do not show discrepancies: the probability of failure and the cost of their Pareto solutions are slightly different. On the other hand, increasing the reliability threshold R, such as in second and third system configurations the discrepancies become more evident.

Even if the random generator method is able to find feasible solutions, in general they are quite different from the solutions returned by the other two methods (see Fig. 9).

Fig. 10 shows the approximate Pareto curves obtained from solving the optimization problem using the three approaches with respect to the second system configuration, where it is required that $T = 15$ and $R = 0.8$. Fig. 11 reports the approximate Pareto curves obtained from solving the optimization problem using the three approaches with respect to the third system configuration, where it is required that $T = 24$ and $R = 0.8$. In the figures we have circumscribed the feasible solutions. Similarly to Table 3 for the first configuration, Tables 4 and 5 report the detailed results of the experimentation for the second and third system configuration, respectively.

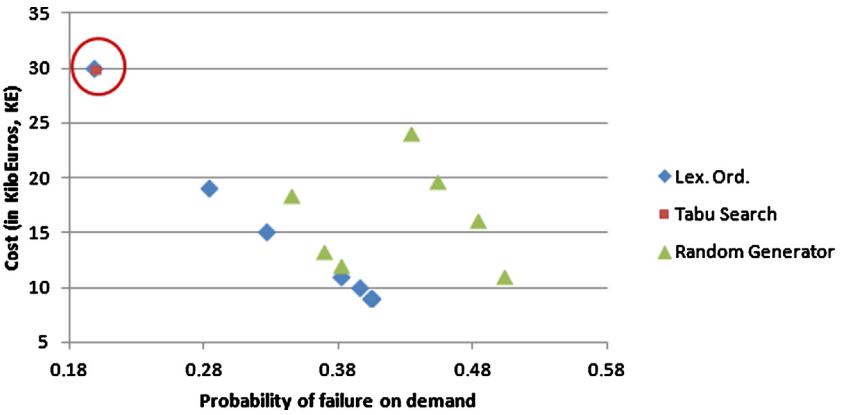


Fig. 10. Comparison of the three approaches with respect to the second configuration.

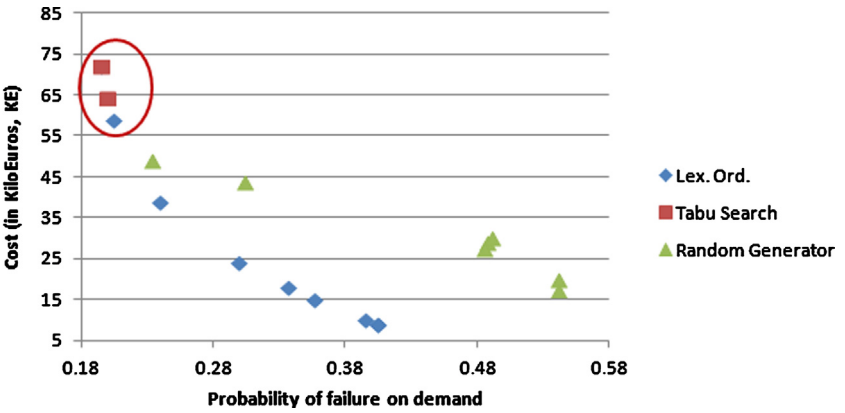


Fig. 11. Comparison of the three approaches with respect to the third configuration.

Table 4
Results from Lexicographic, Tabu Search, and Random Generator for the second configuration.

Lexicographic	Tabu Search	Random Generator
$[(C_{10}, 1), (C_{20}, 1), (C_{30}, 1)]$ $FRsys = 0.404862$ $Cost = 9.15$ KE	$[(C_{10}, 277), C_{21}, C_{31}]$ $FRsys = 0.199833$ $Cost = 29.88$ KE $Time = 11$ sec, $\alpha = 1, 0.2, 0.01, 0$	$[C_{12}, (C_{20}, 3)C_{32}]$ $FRsys = 0.4532485$ $Cost = 19.65030$ KE
$[(C_{10}, 3), (C_{20}, 0), (C_{30}, 0)]$ $FRsys = 0.404199$ $Cost = 9.15$ KE		$[(C_{10}, 2), (C_{20}, 7)C_{32}]$ $FRsys = 0.344271$ $Cost = 18.450$ KE
$[(C_{10}, 23), (C_{20}, 0), (C_{30}, 0)]$ $FRsys = 0.395788$ $Cost = 10.15$ KE, $\epsilon = 0.999$		$[C_{11}, C_{21}, C_{32}]$ $FRsys = 0.4333424$ $Cost = 24$ KE
$[(C_{10}, 3), C_{22}, (C_{30}, 0)]$ $FRsys = 0.382359$ $Cost = 11.15$ KE, $\epsilon = 2$		$[(C_{10}, 0), C_{22}, (C_{30}, 20)]$ $FRsys = 0.380782$ $Cost = 12.002$ KE
$[(C_{10}, 23), (C_{20}, 0), C_{31}]$ $FRsys = 0.326879$ $Cost = 15.15$ KE, $\epsilon = 6$		$[(C_{10}, 29), C_{22}, (C_{30}, 16)]$ $FRsys = 0.368689$ $Cost = 13.254$ KE
$[(C_{10}, 63), C_{21}, C_{31}]$ $FRsys = 0.283528$ $Cost = 19.16$ KE, $\epsilon = 10$		$[C_{12}, (C_{20}, 0), (C_{30}, 10)]$ $FRsys = 0.502234$ $Cost = 11.001$ KE
$[(C_{10}, 280), C_{21}, C_{31}]$ $FRsys = 0.198841$ $Cost = 30.03$ KE, $\epsilon = 23$		$[C_{11}, C_{21}, (C_{30}, 23)]$ $FRsys = 0.482429$ $Cost = 16.152$ KE

Table 5
Results from Lexicographic, Tabu Search, and Random Generator for the third configuration.

Lexicographic	Tabu Search	Random Generator
$[(C_{10}, 1), (C_{20}, 1), (C_{30}, 1)]$ $FRsys = 0.404862$ $Cost = 9.15$ KE	$[(C_{10}, 460), (C_{20}, 267), (C_{30}, 380)]$ $FRsys = 0.199966$ $Cost = 64.46$ KE $Time = 65$ sec, $\alpha = 1, 0.2$	$[C_{11}, C_{21}, (C_{30}, 283)]$ $FRsys = 0.487346$ $Cost = 29.178$ KE
$[(C_{10}, 3), (C_{20}, 0), (C_{30}, 0)]$ $FRsys = 0.404199$ $Cost = 9.15$ KE	$[(C_{10}, 460), (C_{20}, 420), (C_{30}, 380)]$ $FRsys = 0.195004$ $Cost = 72.13$ KE $Time = 65.725$ sec, $\alpha = 0$	$[C_{12}, (C_{20}, 381), (C_{30}, 13)]$ $FRsys = 0.491148$ $Cost = 30.239$ KE
$[(C_{10}, 23), (C_{20}, 0), (C_{30}, 0)]$ $FRsys = 0.395788$ $Cost = 10.15$ KE, $\epsilon = 0.999$		$[C_{11}, C_{22}, C_{31}]$ $FRsys = 0.541594$ $Cost = 20$ KE
$[(C_{10}, 123), (C_{20}, 0), (C_{30}, 0)]$ $FRsys = 0.356958$ $Cost = 15.16$ KE, $\epsilon = 6$		$[C_{12}, C_{21}, C_{31}]$ $FRsys = 0.541594$ $Cost = 17.500$ KE
$[(C_{10}, 183), C_{20}, C_{30}]$ $FRsys = 0.336165$ $Cost = 18.13$ KE, $\epsilon = 9$		$[(C_{10}, 483), C_{22}, (C_{30}, 277)]$ $FRsys = 0.233585$ $Cost = 49.076$ KE
$[(C_{10}, 303), (C_{20}, 0), (C_{30}, 0)]$ $FRsys = 0.299842$ $Cost = 24.18$ KE, $\epsilon = 15$		$[(C_{10}, 442), (C_{20}, 80), C_{32}]$ $FRsys = 0.304315$ $Cost = 44.152$ KE
$[(C_{10}, 460), (C_{20}, 0), (C_{30}, 143)]$ $FRsys = 0.239423$ $Cost = 39.21$ KE, $\epsilon = 30$		$[C_{12}, C_{21}, (C_{30}, 303)]$ $FRsys = 0.485975$ $Cost = 27.680$ KE
$[(C_{10}, 460), (C_{20}, 163), (C_{30}, 380)]$ $FRsys = 0.204292$ $Cost = 59.25$ KE, $\epsilon = 50$		
$[(C_{10}, 460), (C_{20}, 420), (C_{30}, 380)]$ $FRsys = 0.195004$ $Cost = 72.13$ KE, $\epsilon = 100$		

The tabu search, even when the search space became more complex, has returned feasible solutions in a short time: its execution time increased from few seconds (about eleven seconds) to few minutes (about one minute). On the opposite, the lexicographic method has taken more time while increasing the search space. Finally, the random solution generation has also used a short computation time (usually not finding feasible solutions) by making its decisions randomly.

Discussion on the compared approaches. The comparison of the results has revealed that the lexicographic method outperforms the choices of the random generator solution generator. In fact, the lexicographic finds optimal solution with a short time while increasing the search space complexity with respect to the random generator.[9] The tabu search, in turn, outperforms the random generator and the lexicographic method. For example, the tabu algorithm allows tackling well-known drawback such as the specification of preferences to arrange the objective functions in order of importance. In fact, it may be difficult to specify preferences with no/limited knowledge on the optimal objective values and, as a consequence, the Pareto-optimal solutions consistent with the given preferences could present the effects of wrong choices. A detailed discussion on the method scalability is carried out in Section 7.

6.3. Application of reliability tactics to a tabu search solution

In order to improve the quality attributes of the current solution – the one obtained by the tabu search (as first process iteration) from the third system configuration – some tactics can be taken in consideration and properly composed. We here show the application of a reliability tactic to the STS architecture. Let us assume for the STS the following extra-functional requirement (taken from [36]):

NFR1. *The STS reliability should be greater than 0.85*.

To address NFR1, reliability (or availability) tactics for recognizing faults and recovery can be adopted (see Appendix A.1 for a description of reliability tactics). Tactics for *Fault Detection*, for example, can be composed with the tactics for *Recovery Reintroduction* and for *Recovery-Preparation and Repair*. The tactics for *Fault Detection* allow the detection and the notification of a fault to a monitoring component or to the system administrator. The tactics for *Recovery Reintroduction* allow the restoring of the state of a failed component, whereas the tactics for *Recovery Preparation and Repair* are for the recovering and repairing of a component from a failure.

[9] Note that in the figures of results we report Pareto solutions for the lexicographic method and the tabu search. We have discarded Pareto-dominated solutions.

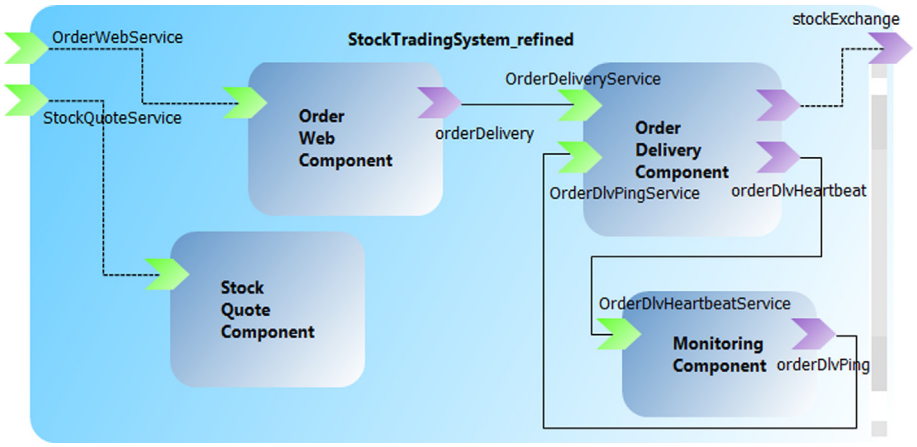


Fig. 12. Adapting the STS by applying tactics for NFR1.

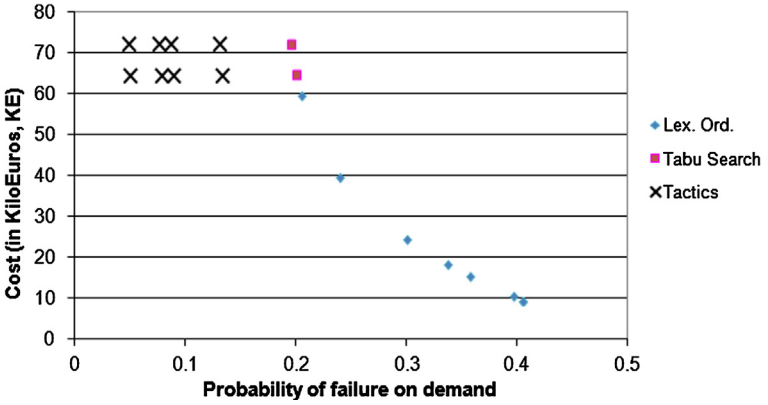


Fig. 13. Comparison of the two approaches and the tactics with respect to the third configuration.

Specifically, we support NFR1 by combining the tactics *Ping/Echo* and *Heartbeat* (tactics for *Fault Detection*), and the tactic *State Resynchronization* (a tactic for *Recovery Reintroduction*). In the tactic *Ping/Echo*, one monitoring component issues a ping and expects to receive back an echo, within a predefined time, from the component under scrutiny; if such a message is not received within the time limit, then it considers that component to be in failure mode, and takes corrective actions. In the tactic *Heartbeat*, one component emits a heartbeat message periodically and another component listens for it; if the heartbeat fails, the originating component is assumed to have failed and a fault correction component is notified. The tactic *State Resynchronization* restores (through a resynchronization manager) the state of a source component through resynchronization with the state of a backup component, either when the state of the source component is changed or when the source component is recovered from a failure.

The resulting assembly SCA obtained by applying the reliability tactics mentioned above is shown in Fig. 12. It contains a new composite component `MonitoringComponent` for the fault detection tactics *Ping/Echo* and *Heartbeat*. Of course, this adaptation implies a change of the components shape (i.e., in the required/provided interfaces) and also of their behavior. The tactic *State Resynchronization* involves concepts of a state resynchronization manager, source components and backup components; but, these concepts are non-visible at architectural level of the top SCA assembly.

The (implementation of one) combination of such reliability tactics can be specified by using basic parameters of the adaptation process, e.g., the reliability of a component depends on the redundant components used by the tactics for it, and be a measure of developer skills (e.g., two applications of the same tactics may impact differently on the in-house instances parameters).

As shown in Fig. 13 for the third configuration of the STS, the application of reliability tactics to the solutions provided by the tabu search may increase the system reliability. As done in [54], we have formulated, the probability of failure on demand of an in-house developed instance as a function of the probability that the instance is faulty, the testability and the number of successful test cases performed. The probability that an instance is faulty is an intrinsic property of the instance that depends on its internal complexity. The more complex the internal dynamics of the component instance is, the higher is the probability that a bug has been introduced during its development. The testability expresses the conditional probability that a single execution of a software fails on a test case following a certain input distribution [55].

1. Model Parameters and Variables

n	number of system components
$\underline{J}_i$	COTS instances available for the component i
$\bar{J}_i$	in-house developed instances available for the component i
s_i	average number of invocations
μ_{ij}	probability of failure on demand of the COTS instance j
p_{ij}	probability that the in-house instance j is faulty
π_{ij}	testability of the in-house instance j
x_{ij}	COTS instance j (in-house instance) selected for the component i
N_{ij}^{tot}	number of tests performed on the in-house j developed for the component i
N_{ij}^{suc}	number of of successful (i.e. failure-free) tests performed on the in-house j
$N_{ij}^{suc} = (1-\pi_{ij})N_{ij}^{tot}$	

2 . System Probability of failure on demand: $1 - \text{RelSys}$

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

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

5. The testability of the j-th in-house developed instance: $\pi_{ij} = P(failure|\ prob.\ distribution\ of\ input)$

Fig. 14. Reliability Model used in [54].

Table 6
Tactics Application Results for the solution of cost 64.46 KE.

$p_{10} = 0.5,\ p_{20} = 0.4,$ $p_{30} = 0.4$	$p_{10} = 0.3,\ p_{20} = 0.4,$ $p_{30} = 0.4$	$p_{10} = 0.3,\ p_{20} = 0.3,$ $p_{30} = 0.3$	$p_{10} = 0.2,\ p_{20} = 0.2,$ $p_{30} = 0.2$
$Rsys = 0.867397$	$Rsys = 0.911908$	$Rsys = 0.922630$	$Rsys = 0.950988$

Table 7
Tactics Application Results for the solution of cost 72.13 KE.

$p_{10} = 0.5,\ p_{20} = 0.4,$ $p_{30} = 0.4$	$p_{10} = 0.3,\ p_{20} = 0.4,$ $p_{30} = 0.4$	$p_{10} = 0.3,\ p_{20} = 0.3,$ $p_{30} = 0.3$	$p_{10} = 0.2,\ p_{20} = 0.2,$ $p_{30} = 0.2$
$Rsys = 0.869397$	$Rsys = 0.914010$	$Rsys = 0.924281$	$Rsys = 0.952146$

Fig. 14 reports the reliability model used in [54]. In particular, the figure shows how the probability to be faulty and the testability are combined in order to predict the probability of failure on demand on an in-house developed component. In [54], an optimization model is formulated, in order to support the decision whether to buy software components or to build them in-house upon designing a software architecture. The selection criterion is based on cost minimization of the whole assembly subject to constraints on system reliability and delivery time. In addition, the model suggests the amount of unit testing to be performed on each in-house developed component.

In [54] procedures to estimate reliability parameters are suggested. Obviously, different (implementation of) combination reliability tactics may impact differently on the in-house instance parameters.

Experimental results. Tables 6 and 7 report the detailed results of the reliability tactics application to the solutions provided by the tabu search (with respect to the third system configuration). The tables are organized as follows: each column represents a value of the probability that the in-house instances are faulty p_{i0}, which we have estimated after tactic application, each entry (row, column) represents the system reliability resulting after the tactics application.

By analyzing the results we can observe that while varying the tactics application the solution with cost 72.13 KE becomes Pareto-dominated by the solution with cost 64.46 KE (i.e., the reliability of the solution with cost 64.46 KE becomes slightly different to the one of solution with cost 72.13 KE). Therefore, the application of design solutions may decrease the cost to adapt the system. This highlights the novelty and capabilities of our approach. In fact, this difference would have not been perceived by using approaches that do not predict the quality attributes (and the adaptation cost) of the system resulting after the design solutions application (like, for example, the works in [3,21] and [56]).

On the other hand, the application of more sophisticated adaptation actions (e.g., the ones of tactics) may require a higher adaptation cost. This cost could be required, for example, to introduce new components required by the tactics (e.g., the *Recovery* tactics application may involves concepts of clients, a primary component, backup components and a state

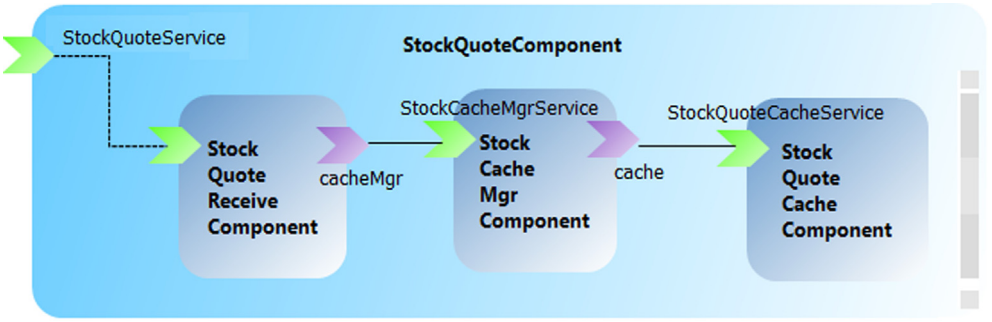


Fig. 15. The StockQuote composite for NFR1 and NFR2 tactics.

resynchronization manager). These new components can likely increase the average time required to perform a test case on an in-house instance. For example, if the average time required to perform a test case of all in-house components increases from 0.05 to 0.1, the adaptation cost would increase to 135.25 KE and 119.92 KE, respectively, from 72.13 KE and 64.46 KE.

Applying again the tabu search. After the tactics application, if the tabu search is applied (as second iteration process) again then a better candidate solution could be found. For example, if after the tactics application, that involves $p_{10} = 0.2$, $p_{20} = 0.2$ and $p_{30} = 0.2$, we use the tabu search the following candidates are returned.

First candidate: $[(C_{10}, 90), (C_{20}, 67), (C_{30}, 76)]$ The system reliability is equal to 0.913413 and the cost is equal to 20.67 KE. Note that such a solution involves a lower adaptation cost with respect to the one of solutions obtained only with the tactics application (see Table 6 and 7).

Second candidate: $[(C_{10}, 470), C_{23}, (C_{30}, 500)]$ The system reliability is equal to 0.902940 and the cost is equal to 60.60 KE. Note that in this case the tabu search, other than changing the number of test, selects the COTS instance C_{23} for the second component.

6.4. Application of reliability and performance tactics

The adoption of specific design solutions (like tactics) for a quality attribute is often dictated by the trade-off with other quality attributes. Since tactics suggest different adaptations actions, they may differ for adaptation cost and/or for the system quality they achieve. Our optimization process allows the combination of different tactics by predicting the resulting system quality. We here show, for example, how to compose reliability and performance tactics for the STS. Let us assume for the STS the further performance requirement[10]:

NFR2*. The trading information of about 600 items is sent every second on average to the STS. Updating such a high volume of information imposes an intensive load on the STS's database, which may cause slow performance. In order to minimize the impairment of performance, updates should be the least possible.*

This new requirement is concerned with the performance of the STS since when updates are received at such a high frequency, some items (e.g., stocks having less trades) may not have changes in every update. To this purpose, many systems usually use local caches to filter out the actual items that need to be updated by comparing the received update with the previous update. Instead of having all consumption points (clients) bound to one data repository source, local copies significantly improve performance of the overall system by reducing contention. To satisfy such new requirement, therefore, in addition to the reliability tactics (*Ping/Echo*, *Heartbeat*, and *State Resynchronization*) applied previously to support NFR1, we adopt the tactic *Maintain Multiple Copies* (a tactic for *Resource Management*, see Appendix A.2 for a detailed description) to force a caching mechanism. Such a tactic introduces: a data repository of existing data, a cache maintaining local copies of frequently requested data, and a cache manager that is responsible to keep local copies consistent and synchronized.

The resulting SCA assembly obtained by applying all the tactics mentioned above is still the one shown in Fig. 12, but the `StockQuoteComponent` is refined into a composite component (see Fig. 15) to support selective updates through the tactic *Maintain Multiple Copies*. The composite `StockQuoteComponent` contains three components, namely a cache `StockQuoteChaceComponent`, a cache client `StockQuoteReceiveComponent`, and a cache manager `StockCache-MgrComponent`. The cache client receives the updates from the external Stock Exchange system, and then it requests the cache manager to update the received update. The cache manager looks up the previous update in the cache and compare it with the one that is received and identify the items that have changes. Only those items that have changes are updated in the data repository, thus reducing the repository update load [36].

Below, we provide the experimental results obtained by the optimization process when we apply first the tactic for NFR2, and then the tactics for NFR1 to a candidate solution obtained from a previous Tabu Search application.

Application of tactics to a tabu search solution. Starting from the candidate $[(C_{10}, 460), (C_{20}, 420), (C_{30}, 380)]$ (with a cost equal to 72.13 KE and system reliability equal to 0.804996) returned by the Tabu Search for the third system configu-

[10] Such a requirement corresponds to the requirement NFR3 of the case study in [36].

ration, let us apply the *Maintain Multiple Copies* tactic for NFR2. Such a tactic involves adaptation actions on the *Stock Quote Component* (i.e., C_2). If to implement the tactic (i.e., a cache, a cache client and a cache manager) the unitary development cost of the in-house instance C_{20} increases from 3 to 5, and its testability increases from 0.002 to 0.006, the reliability of the solution (predicted using the reliability model used in [54]) increases from 0.804996 to 0.811168, and its cost increases from 72.13 KE to 74.21 KE. This is due to the fact that, once fixed the number of test cases successfully performed, the probability of failure on demand of a component decreases while increasing its testability. Thus, let us combine the *Maintain Multiple Copies* tactic with the tactics for NFR1. Let us consider an application of such performance and reliability tactics, that increases the average time required to perform a test case for C_{10}, C_{20} and C_{30} from 0.05 to 0.1, from 0.05 to 0.2 and from 0.05 to 0.1, respectively. In the following, we report examples of candidates generated for different values of the unitary development cost of the in-house components.

- *First Candidate*: If the unitary development cost of C_{10}, C_{20} and C_{30}, increases from 1 to 2, from 5 to 6 and from 5 to 6, respectively, and the probability that the instance is faulty is equal to 0.4, 0.4 and 0.3, then the system reliability increases to 0.903389 and the cost increases to 182.68 KE.
- *Second Candidate*: If the unitary development cost of C_{10}, C_{20} and C_{30}, increases from 1 to 3, from 5 to 7 and from 5 to 9, respectively, and the probability that the instance is faulty is equal to 0.3, 0.2 and 0.2, then the reliability increases to 0.934843 and the cost increases to 187.67 KE.
- *Third Candidate*: If the unitary development cost of C_{10}, C_{20}, C_{30} increases from 1 to 2, from 5 to 6 and do not change, respectively, and the faulty probability is equal to 0.4, 0.4 and 0.5, then the system reliability increases to 0.887101 and the cost increases to 181.68 KE.

Applying again the Tabu Search. If we apply again the metaheuristic technique, then better solutions can be found. For example, if we use the Tabu Search by starting from the *third candidate*, then we obtain the following candidate: $[(C_{10}, 800), C_{23}, (C_{30}, 700)]$. The reliability is equal to 0.881299 and the cost decreases from 181.68 KE to 163.30 KE. Such solutions differ for the number of test for C_{10} and C_{30}, and the component selected for C_2.

6.5. Combining tactics and architectural patterns

Let us assume for the STS the following new functional requirement:

F1. *The STS should convert the stock price in the user preferred currency*.

To address *F1* we adopt the architectural pattern *Pipe and Filter*. This pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters [57]. Specifically, in the STS architecture a filter is added to the `OrderDeliveryComponent` for converting the currency into the required one, while the users check the current price of stocks, place buy or sell orders and review traded stock volume. Similarly, a filter is added to the `StockQuoteComponent` for converting the currency into the required one when the users want to know stock quote information.

Tactics may impact on architectural patterns (see, e.g., [3,21] where a qualitative analysis of the interaction between reliability tactics and architectural pattern is provided). Let us assume, for example, that for the STS, other than F1, the following non-functional requirement is required:

NFR1*: *The STS reliability should be greater than 0.9*.

To address both F1 and NFR1* we compose (as suggested in [3]) the *Pipe and Filter* architectural pattern with the previous reliability tactics for *Fault Detection* and a reliability tactic for *Recovery-Preparation and Repair*. This last is the tactic *Voting*[11]: processes running on redundant processors each take equivalent input and compute a simple output value that is sent to a voter; if the voter detects deviant behavior from a single processor, it fails it.

Starting from the candidate $[(C_{10}, 460), (C_{20}, 420), (C_{30}, 380)]$ (with a cost equal to 72.13 KE and system reliability equal to 0.804996) returned by the Tabu Search for the third system configuration (see Fig. 7 and Section 6.2), let us apply the *Pipe and Filter* and the reliability tactics. Let us consider an implementation, where the average time required to perform a test case on C_{10}, C_{20} and C_{30}, increases from 0.05 to 0.1, from 0.05 to 0.2 and from 0.05 to 0.1, respectively, and the unitary development time of the instances increases from 1 to 3, from 3 to 6 and does not change. Besides, the probability that C_{10}, C_{20} and C_{30} are faulty results equal to 0.3, 0.2 and 0.2, respectively. NFR1* is not satisfied with respect to these design pattern and tactics. In fact, the system reliability is equal to 0.863466, while the adaptation cost is equal to 182.68 KE.

Applying again the Tabu Search. If we would apply again the Tabu Search algorithm by generating new candidates also using user adaptation plans, after the tactic and design pattern application, the following candidate is returned: $[(C_{13}), (C_{20}, 420), (C_{30}, 380)]$ with reliability equals to 0.954962. Note that NFR1* is satisfied. Besides, the cost decreases from 182.68 KE to 139.58 KE. The Tabu Search returns such a solution by applying a user adaptation plan.

This result highlights the capabilities of our approach with respect to, for example, the works in [3,21,56]. In fact, this difference is not perceived by using approaches that do not combine the metaheuristic techniques and the design solutions, and do not predict the quality attributes of the system resulting after the adaptation actions application.

[11] As alternative also the tactic *Active Redundancy* can be used (see Appendix A.1 for a description).

Table 8
Parameters of instances available for existing components.

	Instance alternatives	Cost c_{ij}	Average delivery time d_{ij}	Average no. of invocations s_i	Prob. of fail. on demand μ_{ij}
C_1	C_{11}	1	4	170	0.0001
	C_{12}	3	4	170	0.00004
C_2	C_{21}	3	4	20	0.0003
	C_{22}	5	5	20	0.0004
C_3	C_{31}	2	4	20	0.00001
	C_{32}	3	13	20	0.000002
C_4	C_{41}	1	3	45	0.00003
	C_{42}	3	2	45	0.000004
C_5	C_{51}	9	4	50	0.00003
	C_{52}	11	8	50	0.000001
C_6	C_{61}	8	4	10	0.00002
	C_{62}	10	3	10	0.00001
C_7	C_{71}	7	2	5	0.00025
	C_{72}	9	3	5	0.00001

Table 9
Parameters for in-house developed instances.

	Development time t_{i0}	Testing time τ_{i0}	Unitary development cost $\bar{c}_{i0}$	Average no. of invocations s_i	Testability π_{i0}
C_1	1	1	1	170	0.001
C_2	3	1	1	20	0.002
C_3	5	1	1	20	0.001
C_4	1	1	1	45	0.002
C_5	1	1	1	50	0.003
C_6	2	1	1	10	0.002
C_7	1	1	2	5	0.0001

7. Numerical evaluation

To show the effectiveness of the proposed approach we here illustrate the results we have obtained performing an extensive evaluation. Specifically, we analyze the optimization process sensitivity to changes in its input parameters (e.g., size of the possible components). For this experimental study, we used a system containing seven software components.

We assume that several instances (i.e., implementations) of an existing component may be available on the market, all *equivalent* from the functional viewpoint. Basically, the instances differ each other for costs and non-functional properties such as reliability and response time. Table 8 shows the initial values that we have considered for the parameters of SCA components available on the market. Similarly, Table 9 shows the initial parameters that we have considered for in-house SCA components.

The second column of Table 8 lists, for each component, the set of instance alternatives available. For each alternative: the adaptation cost c_{ij} (in KiloEuros, KE) is given in the third column, the average delivery time d_{ij} (in time unit) is given in the fourth column, the average number of invocations of the component in the system s_i is given in the fifth column, finally the probability of failure on demand μ_{ij} is given in the sixth column.

For each component in Table 9: the estimated development time t_{i0} (in time unit) is given in the third column and the average time required to perform a test case τ_{i0} (in time unit) is given in the fourth column, the unitary development cost $\bar{c}_{i0}$ (in KE per day) is given in the fifth column, the average number of invocations s_i is given in the sixth column, and finally the component testability π_{i0} is given in the last column.

Application of the TS algorithm. In order to demonstrate the scalability of the TS algorithm, we solve randomly generated system configurations. Specifically, starting from the initial values of the parameters, we have generated nine different system configurations (here also called *perturbed configurations*) by randomly changing the parameters. For example, two of these configurations differ both for the COTS instances' parameters (as we explain below) and for the in-house instances' parameters, which we have slightly decreased/increased. For each perturbed configuration, we have generated six COTS component bases. For each existing component, the number of COTS components spans from 6 to 60 (i.e., related to one of the six component bases). We have generated these new COTS components by randomly perturbing the initial COTS instances available for each component (see Table 8). For a component base, the number of COTS instances available for each component does not change across the perturbed configurations, but each configuration's base is based on a different set of component parameters.

The perturbed configurations' parameters and the ones of the component bases have been varied for showing the behavior of the TS while increasing the search space complexity. For example, two COTS instances available for the same

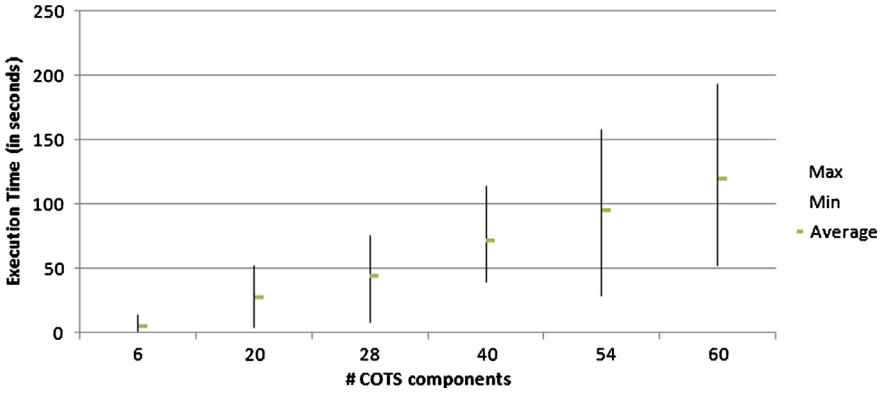


Fig. 16. Execution time of the TS algorithm.

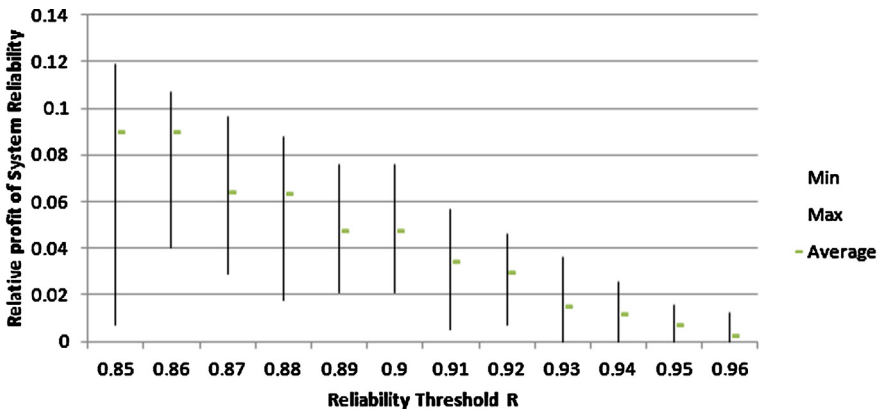


Fig. 17. Relative Profit of the System Reliability.

component, differ for the probability of failures values. These values differ in at most 80% of the probability of failure of the initial COTS components. For a given perturbed configuration, the component bases differ for the probability to find a Pareto solution (or end the TS process): the first base, characterized by a lower number of COTS instances, has a higher probability with respect to the other ones, characterized by a set of selected components more complex to be analyzed.

For each perturbed configuration, we have solved the TS algorithm by replacing the components by buying or building components basing on cost and non-functional factors (see Section 6.1), for a set of reliability values bound to R. This latter spans from 0.85 from 0.96. Specifically, given a perturbed configuration, we have used the TS algorithm under these different reliability constraints while varying the configuration' component bases.

In Fig. 16 we illustrate the performance of the TS algorithm in terms of COTS component set size. The figure shows the results for the nine perturbed configurations with $6, 20, \ldots, 60$ COTS components. For the sake of continuity, we have imposed the same parameters for the example considered in the previous section (see Section 5.1), i.e., we imposed a number of interactions of 50, and the tabu list length limit of 45 to each experiment.

The bars refer to the execution of the TS for the perturbed configurations for every component bases' parameter setting. In particular, each bar – corresponding to one component base – contains the higher, lower and the average execution time of the TS.

As expected, the lower (higher or average) TS execution time increases while increasing the number of COTS components (i.e., while varying the component sets). For example, with a component base of 28 components per each component, the TS average execution time is about 45.91 sec, whereas if the number of component increases to 40, then the TS average execution time is about 73.2 sec. Even for higher number of components, the increase is still acceptable (about to 120 sec for 60 COTS components for each component).

The figure indicates that the tabu search manages the search space, even when it became more complex: its execution time increased from few seconds (about six seconds in average) to few minutes (about two minutes in average).

In Fig. 17 we illustrate the results of the TS in terms of the system reliability. The figure shows the gain in system reliability obtained with the TS results. The x-axis represents the variation of the reliability threshold (R).

The figure shows the results for the nine perturbed configurations with their base of 6 COTS components per each component. Specifically, each bar – corresponding to one reliability threshold's value – contains the higher, lower and the

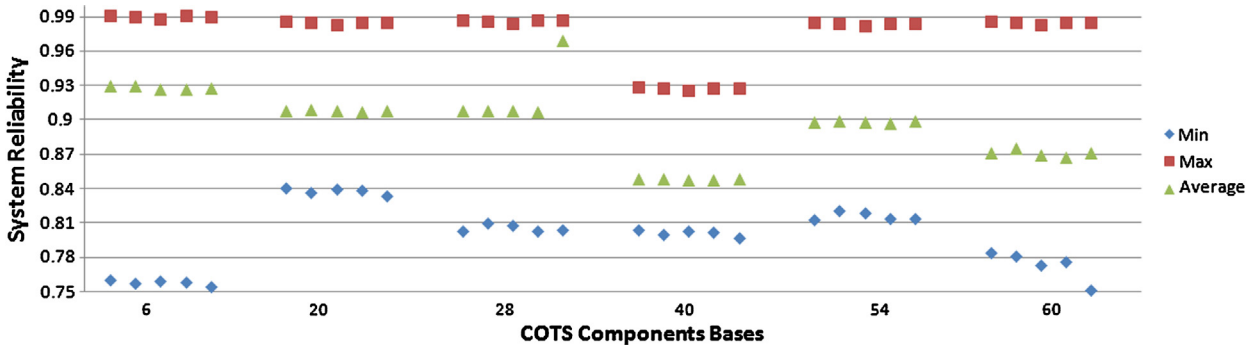


Fig. 18. System Reliability achieved after the application of tactics.

average relative profit[12] obtained for the system reliability. The "failures" of the TS (i.e., it was not able to find feasible solutions for a perturbed configuration) correspond in Fig. 17 to the points of the bars where the lower reliability profit on the y-axis equals 0. Since TS algorithm leverages the *TS Initial Solution Generation* method, that generates a set of feasible solutions (see Section 5.1), in these TS failure cases, there are not solutions that fulfill the quality constraints. To address such a problem, the optimization process applies existing design solutions. However, other different strategies could also be used (e.g., an algorithm combining heuristics, local search, user adaptation plans, service selection and re-deployment actions). We consider this to be an interesting avenue of future research.

As expected, the lower (higher or average) gain in system reliability decreases while increasing the reliability threshold (R). For example, with R equals to 0.9, the average profit is about 0.048, whereas if R increases to 0.95, then the average profit decreases, and it is about 0.0007.

Application of reliability and performance tactics to the TS solutions. In order to analyze the tactics application, we apply randomly generated tactics. Specifically, starting from the initial values of the parameters (see Table 8 and Table 9), we have generated five sets of tactics. Each set contains different implementations of reliability and performance tactics (e.g., the combination of the *Ping/Echo*, *Heartbeat*, *State synchronization*, and the *Maintain Multiple Copies* tactics). The numbers of tactics spans from 13 to 70 in five steps (i.e., one for each set).

As remarked in Section 6.3, a tactic implementation is characterized by a particular system parameters setting. On one hand, the combinations of tactics have been randomly varied for analyzing the optimization process sensitivity to changes in parameters. On the other hand, the number of tactics has been increased to show the process' behavior while increasing the search space complexity.

For example, two implementations of the *Maintain Multiple Copies* tactic differ slightly for values of the development cost (see Table 9). These values differ in at most 25% of the development cost of the initial in-house instances. Different implementations of the reliability tactics have been generated by randomly setting parameters like the probability to be faulty and the testability (see Section 6.3).

We have applied the tactics sets for every TS solutions, returned for the nine perturbed configurations, with R equals to 0.85, with respect to the COTS component bases.

In Fig. 18 we report the results obtained from applying the tactics sets for the TS solutions. The figure shows the system reliability, obtained after the tactics applications, for the TS solutions of the nine perturbed configurations with $6, 20, \ldots, 60$ COTS components. The reliability has been predicted by using the reliability model used in [54] (see Fig. 14). This quality analysis results were run with the LINGO 11.0 optimization package.

The diamonds (triangles and rectangles) refer to the execution of the TS for the perturbed configurations for every component bases' parameter setting. Each group of five diamonds, triangles and rectangles corresponds to the application of the five tactics sets to the TS solutions for the nine perturbed configurations with a fixed COTS components base. Specifically, once fixed a tactics set, the diamond, the triangle and the rectangle represent the higher, lower and the average system reliability, respectively, achieved after the application of the set of tactics.

The figure reveals that, depending on the system parameter setting, any combination of tactics may have a considerable impact on the system reliability. For example, with a component base of 20 components per each component, the average system reliability with the first set of tactics (first triangle) is about 0.9, whereas with the application of the same set of tactics, to the TS solutions returned for the set of 60 COTS component, the average system reliability decreases about to 0.87. Therefore, our optimization process aims at quantifying such impact to suggest the best adaptation actions that still minimizes the costs while satisfying the reliability constraints.

On the other hand, our optimization process allows to analyze the right tradeoff among the system software qualities. In fact, depending on the system parameter setting, any combination of tactics may have a considerable impact on differ-

[12] For a feasible solution s returned by the TS, we have estimated the profit as follows: $(RelSys(s) - R)/RelSys(s)$, where $RelSys(s)$ is the system reliability achieved with the solution s.

Fig. 19. Average Execution Time of the TS algorithm and Tactics' combination.

ent competing and conflicting objectives such as reliability, response time, throughput and consumption of resources. For example, with the application of the last set of tactics (i.e., the one with 70 tactics), to the TS solutions returned for the set of 40 COTS component, the performance tactics (e.g., *Maintain Multiple Copies* tactic) improves the system performance (see Section 6.4), whereas, the average system reliability stays under the threshold R (equals to 0.85). In fact, it is equal to 0.849257. This highlights the potential of the optimization process to drive architectural decisions under quality attributes tradeoffs.

Combined application of TS algorithm and tactics. In Fig. 19 we illustrate the performance of our optimization process in terms of the composition of the TS algorithm and the tactics application. The figure shows the results obtained from applying the tactics sets for the TS solutions in terms of system reliability and average execution time, i.e., the one related to the TS execution plus the ones for the tactic applications. Specifically, the figure reports the system reliability, obtained after the tactics applications, for the TS solutions of the nine perturbed configurations with $6, 20, \ldots, 60$ COTS components. The x-axis represents the average execution time per component bases. Each curve represents the average value of the system reliability, obtained after the tactics application to the TS solutions, with respect to the tactics sets. In particular, the curve with diamonds reports the lower average values, whereas the curve with triangles reports the higher average values. For example, after the application of all tactics sets, with the average execution time equals to 9,446 sec (related to the base of six COTS instance per component), the lower average value of the system reliability is equal to 0.927327.

As expected, the average execution time of our optimization process increases while increasing the number of COTS components, i.e., for different COTS component sets with different probability to find a Pareto solution (or end the TS process). For example, with a component base of 20 components, the average execution time is about 32 sec, whereas if the number of component increases to 40, the average execution time increases about to 79 sec. Even for higher number of components and tactics, the increase is still appreciable.

The figure reveals that the tactics application (depending on the time to perform the quality analysis) takes a short computation time (see Fig. 16 for the TS execution time). It was out of the scope of this paper to deal with the tactics implementation effort, but we want to work on this aspect. We are planning to enhance our optimization process with the well-assessed cost/time models (e.g., COCOMO [58] cost model). In fact, the developers of a system could implement a tactics by adopting different strategies of development. Therefore, the values of cost and time of tactics could vary due to the values of the development process parameters (e.g. experience and skills of the developing team).

8. Conclusions and future work

In this article, we presented an automatic optimization process for adaptation space exploration of service-oriented applications. The adaptation is based on trade-offs between functional and extra-functional requirements. The optimization method combines the application of both metaheuristic search techniques and architectural design patterns and tactics. The proposed methodology relies on heterogeneous service assembly and open tools and runtime infrastructures to process architectural models that are directly tight to the real assembled components implementations and their distributed deployment.

Currently, we are implementing a prototype to compare different implementations of our optimization process (e.g., with heuristics depending on application domain or quality attributes) on a certain number of applications. We also intend to compare several metaheuristics, because different techniques might work differently for different application domains (e.g., neighborhood relations, search space, and initial solution generation).

As future work, we intend to enhance our adaptation space exploration methodology towards several directions. We here described a preliminary experimental validation of our approach focusing mainly on static adaptations. We postpone as future work also the experimental validation on dynamic adaptations. In particular, we intend to support the right trade-off between the adaptation overhead (due, e.g., to the frequent execution of the reasoning algorithms in the *New Candidate Generation* phase) and the accrued benefits of changing the system. Finally, we aim at providing means to allow, for example, the automatic composition of design patterns and tactics and their application to explore the adaptation space under certain assumptions. These might include performance requirements or other quality attributes.

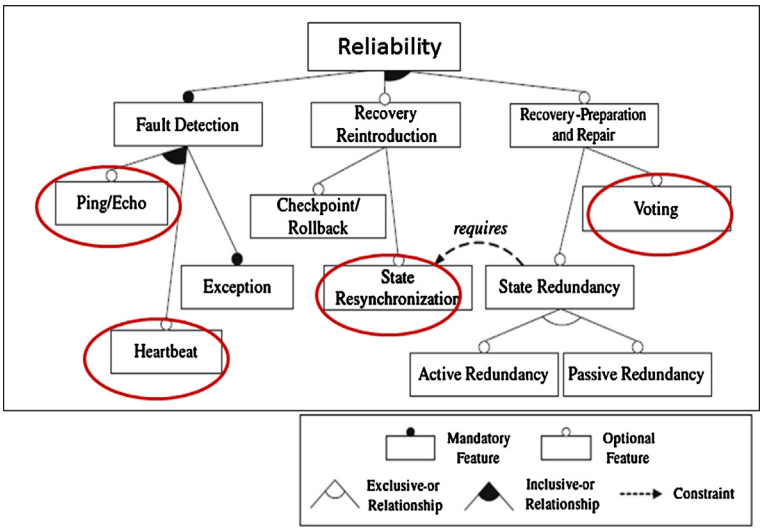


Fig. A.20. Reliability architectural tactic feature model – inspired by the figure in [36] for the availability tactic.

Acknowledgements

We would like to thank the anonymous referees for their comments that helped to substantially improve the quality of the paper. We also are grateful to Vittorio Cortellessa, Fabrizio Marinelli, and Andrea Roli for their interesting discussions on metaheuristic search techniques. This work has been partially supported by the IDEAS-ERC Project SMScom, and by the Italian PRIN project "GenData 2020".

Appendix A. Tactics used in the STS case study

In this section we describe the tactics used in the STS case study by using feature diagrams [59]. In [36] and [60] more details can be found. The formalization of tactics composition is outside the scope of this paper. However, to this extent, the binding roles and the composition roles defined in [36] could be exploited.

A.1. Reliability Tactics

As remarked in [60], it does not exist a universally accepted terminology for the various tactics of fault tolerance.

As shown in Fig. A.20, the several tactics for reliability can be categorized into *Fault Detection*, *Fault Recovery Preparation and Repair*, and *Recovery Reintroduction of a Failed Component* tactics.[13] In Fig. A.20 we have circumscribed the tactics, which we have used.

The *Fault Detection Tactic* is for the detection and notification of a fault to a monitoring component or to the system administrator. The *Recovery Reintroduction of a Failed Component* is for the restoring of the state of a failed component, whereas the *Fault Recovery Preparation and Repair* is for the recovering and repairing of a component from a failure. Each kind of these tactics can be refined into other ones.

Fault Detection Tactic

In the STS case study, we have refined the *Fault Detection Tactic* in *Ping/Echo* and *Heartbeat* tactics – described below.

- *Ping/Echo*: A ping message is sent from a monitoring component to one or more components under scrutiny. The monitoring component expects to receive an echo message back within a predetermined time. If such a message is not received within the time limit, then it considers that the component is in failure mode, and takes corrective actions.
  Therefore, the implementation of such a tactic concerns the creation or the use of a monitoring process, and then all components being watched must be modified to handle the echo messages.
- *Heartbeat*: A heartbeat message is sent from a component at regular interval and a monitoring component listens for it. If such a component does not receive the heartbeat message within a predetermined time, then the originating component is assumed to have failed, and corrective actions are taken.
  Therefore, the implementation of such a tactic involves the creation of a monitoring component, and all components must be modified to send heartbeats at the proper intervals.

[13] The figure is not exhaustive; it could be refined/exploited by considering other reliability tactics.

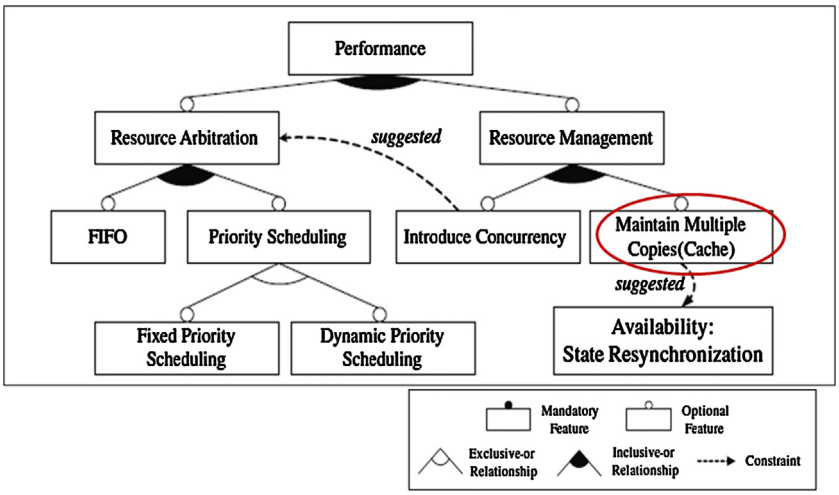


Fig. A.21. Performance architectural tactic feature model [36].

Recovery Reintroduction

In Section 6.4 we have combined the *Fault Detection Tactic* and the *State Resynchronization* tactic (one of the *Recovery Reintroduction* tactics). The scope of this latter tactic is the restoration of the state of a component through resynchronization with the state of a backup component. Thus, the state resynchronization manager, source components, and backup components are the main components involved in such a tactic.

Fault Recovery Preparation and Repair

In Section 6.5 we have combined the *Fault Detection Tactic* and the *Voting* (or *Active Redundancy*) tactic – here below described.

- *Voting*: There are different processes that run on redundant processors. Each process takes equivalent input and computes a single out value. This latter is sent to a voter component that decides which of the results is correct using an algorithm such as majority rules. If each voting component is implemented independently, then the implementation is the strongest. Otherwise, only hardware faults can be detected, while algorithm faults can not. Therefore, the implementation of such a tactic concerns the creation of voter component. Note that if the voting components are running the same software, such a tactic becomes very similar to the *Active Redundancy* tactic – here below detailed.
- *Active Redundancy*: There are redundant components receiving events in parallel – they are always in the same state. When a component fails, the other components can immediately take over.
  The implementation of such a tactic usually involves a central arbitrating component – the redundant components can also perform the arbitrating without a central component.

A.2. Performance Tactics

As shown in Fig. A.21, several tactics for performance can be categorized in *Resource Arbitration* and *Resource Management* tactics. The first is used for improving performance by scheduling requests for expensive resources (e.g., processors, networks), whereas the latter improves performance by managing resources affecting response time.

In Section 6.4 we have used the *Maintain Multiple Copies tactic* (one of the *Resource Management* tactics). This tactic allows the management of resources by keeping replicas of resources on separate repositories, so that contention for resources can be reduced. The implementation issues of such a tactic are related to the clients, a cache – maintaining copies of data that are frequently requested for faster access, a cache manager and a data repository. Thus, if a data request is received, then the cache manager first searches the cache. When the data is not found, the cache manager queries the repository and makes copies of the data into the cache.

References

[1] M. Harman, W.B. Langdon, Y. Jia, D.R. White, A. Arcuri, J.A. Clark, The GISMOE challenge: constructing the pareto program surface using genetic programming to find better programs (keynote paper), in: ASE, 2012, pp. 1–14.
[2] R. Mirandola, P. Potena, Self-adaptation of service based systems based on cost/quality attributes tradeoffs, in: Proc. of SYNACS 2010 – Workshop on Software Services: Frameworks and Platforms, 2010.
[3] N.B. Harrison, P. Avgeriou, How do architecture patterns and tactics interact? A model and annotation, J. Syst. Softw. 83 (10) (2010) 1735–1758.
[4] N. Esfahani, S. Malek, On the role of architectural styles in improving the adaptation support of middleware platforms, in: ECSA, 2010, pp. 433–440.
[5] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni, Model-based performance prediction in software development: A survey, IEEE Trans. Softw. Eng. 5 (2004) 295–310.

[6] F. Rosenberg, M. Müller, P. Leitner, A. Michlmayr, A. Bouguettaya, S. Dustdar, Metaheuristic optimization of large-scale QoS-aware service compositions, in: Proceedings of the 2010 IEEE International Conference on Services Computing, 2010, pp. 97–104.
[7] OSOA, Service Component Architecture (SCA), www.osoa.org.
[8] R. Mirandola, P. Potena, P. Scandurra, An optimization process for adaptation space exploration of service-oriented applications, in: SOSE, IEEE, 2011, pp. 146–151.
[9] A. Bucchiarone, C. Cappiello, E. Di Nitto, R. Kazhamiakin, V. Mazza, M. Pistore, Design for adaptation of service-based applications: main issues and requirements, in: ICSOC/ServiceWave 2009 Workshops, in: LNCS, 2010, pp. 467–476.
[10] M. Salehie, L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Trans. Auton. Adapt. Syst. 4 (2) (May 2009), Article 14.
[11] S.C. Geyik, B.K. Szymanski, P. Zerfos, D.C. Verma, Dynamic composition of services in sensor networks, in: IEEE SCC, 2010, pp. 242–249.
[12] F. Lécué, A. Delteil, A. Léger, O. Boissier, Web service composition as a composition of valid and robust semantic links, Int. J. Coop. Inf. Syst. 18 (01) (2009) 1–62.
[13] J. Xu, Rule-based automatic software performance diagnosis and improvement, Perform. Eval. 67 (8) (2010) 585–611.
[14] T. Parsons, A framework for detecting performance design and deployment antipatterns in component based enterprise systems, in: DSM, ACM, 2005.
[15] V. Cortellessa, A. Martens, R. Reussner, C. Trubiani, A process to effectively identify "Guilty" performance antipatterns, in: FASE, 2010.
[16] L. Grunske, Identifying "good" architectural design alternatives with multi-objective optimization strategies, in: ICSE, ACM, 2006.
[17] A. Martens, H. Koziolek, S. Becker, R. Reussner, Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms, in: WOSP/SIPEW, 2010.
[18] D.A. Menascé, J.M. Ewing, H. Gomaa, S. Malex, J.P. Sousa, A framework for utility-based service oriented design in SASSY, in: ACM DL WOSP/SIPEW 2010 Proc., 2010, pp. 27–36.
[19] R. Kazman, L. Bass, G. Abowd, M. Webb, SAAM: A method for analyzing the properties of software architectures, in: ICSE '94, 1994, pp. 81–90.
[20] L. Dobrica, E. Niemela, A survey on software architecture analysis methods, IEEE Trans. Softw. Eng. 28 (7) (2002) 638–653.
[21] N. Harrison, P. Avgeriou, Implementing reliability: the interaction of requirements, tactics and architecture patterns, in: Architecting Dependable Systems VII, in: Lecture Notes in Computer Science, vol. 6420, 2010, pp. 97–122.
[22] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, D.A. Menascé, Software adaptation patterns for service-oriented architectures, in: SAC, 2010, pp. 462–469.
[23] E. Riccobene, P. Potena, P. Scandurra, Reliability prediction for service component architectures with the SCA-ASM component model, in: V. Cortellessa, H. Muccini, O. Demirörs (Eds.), EUROMICRO-SEAA, IEEE Computer Society, 2012, pp. 125–132.
[24] Apache Tuscany, http://tuscany.apache.org/, 2010.
[25] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, J.-B. Stefani, A component-based middleware platform for reconfigurable service-oriented architectures, Softw. Pract. Exp. 42 (5) (2012) 559–583.
[26] E. Börger, R. Stärk, Abstract State Machines: A Method for High-Level System Design and Analysis, Springer-Verlag, 2003.
[27] The ASMETA toolset website, http://asmeta.sf.net/, 2006.
[28] P. Arcaini, A. Gargantini, E. Riccobene, P. Scandurra, A model-driven process for engineering a toolset for a formal method, Softw. Pract. Exp. 41 (2) (2011) 155–166.
[29] The SCA-ASM design framework [Online], available: https://asmeta.svn.sf.net/svnroot/asmeta/code/experimental/SCAASM.
[30] D. Brugali, L. Gherardi, E. Riccobene, P. Scandurra, Coordinated execution of heterogeneous service-oriented components by Abstract State Machines, in: FACS, in: Lecture Notes in Computer Science, vol. 7253, Springer, Berlin, Heidelberg, 2011, pp. 331–349.
[31] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, J. Stefani, Reconfigurable SCA applications with the frascati platform, in: Proc. of International Conference on Services Computing, IEEE, 2009, pp. 268–275.
[32] A. Martens, H. Koziolek, Performance-oriented design space exploration, in: Proc. of WCOP'08, 2008.
[33] V. Cortellessa, F. Marinelli, P. Potena, An optimization framework for "build-or-buy" decisions in software architecture, Comput. Oper. Res. 35 (10) (2008) 3090–3106.
[34] C. Blum, A. Roli, Metaheuristics in combinatorial optimization: Overview and conceptual comparison, ACM Comput. Surv. 35 (3) (2003) 268–308.
[35] A. Martens, H. Koziolek, Automatic, model-based software performance improvement for component-based software designs, Electron. Notes Theor. Comput. Sci. 253 (1) (2009) 77–93.
[36] S. Kim, D. Kim, L. Lu, S. Park, Quality-driven architecture development using architectural tactics, J. Syst. Softw. 8 (2009) 1211–1231.
[37] P. Arcaini, A. Gargantini, E. Riccobene, P. Scandurra, A model-driven process for engineering a toolset for a formal method, Softw. Pract. Exp. 41 (2) (2011) 155–166.
[38] A. Gargantini, E. Riccobene, P. Scandurra, A metamodel-based language and a simulation engine for abstract state machines, J. Univers. Comput. Sci. 14 (12) (2008) 1949–1983.
[39] A. Carioni, A. Gargantini, E. Riccobene, P. Scandurra, A scenario-based validation language for asms, in: Proceedings of the 1st International Conference on Abstract State Machines, B and Z, ABZ '08, Springer-Verlag, 2008, pp. 71–84.
[40] A. Carioni, A. Gargantini, E. Riccobene, P. Scandurra, Model-driven system validation by scenarios, in: Languages for Embedded Systems and Their Applications (Best of FDL'08), in: Lecture Notes in Electrical Engineering, Springer-Verlag, 2009, pp. 57–69.
[41] A. Gargantini, E. Riccobene, ASM-based testing: coverage criteria and automatic test sequence generation, J. Univers. Comput. Sci. 7 (2001) 262–265.
[42] P. Arcaini, A. Gargantini, E. Riccobene, AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications, in: Proc. of the Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, in: Lecture Notes in Computer Science, vol. 5977, Springer, 2010, pp. 61–74.
[43] C. Attiogbé, P. André, G. Ardourel, Checking component composability, in: W. Löwe, M. Südholt (Eds.), Software Composition, in: LNCS, 2006, pp. 18–33.
[44] M. Marzolla, The `qnetworks` toolbox: A software package for queueing networks analysis, in: K. Al-Begain, D. Fiems, W.J. Knottenbelt (Eds.), Analytical and Stochastic Modeling Techniques and Applications, 17th International Conference, Proceedings, ASMTA 2010, Cardiff, UK, in: Lecture Notes in Computer Science, vol. 6148, Springer, 2010, pp. 102–116.
[45] G. Franks, P. Maly, M. Woodside, D.C. Petriu, A. Hubbard, Layered Queueing Network Solver and Simulator User Manual, LQN Software Documentation, 2006.
[46] I. Epifani, C. Ghezzi, R. Mirandola, G. Tamburrelli, Model evolution by run-time parameter adaptation, in: Proc. of ICSE'09, 2009, pp. 111–121.
[47] Y. Censor, Pareto optimality in multiobjective problems, Appl. Math. Optim. 4 (1977) 41–59.
[48] R. Marler, J. Arora, Survey of multi-objective optimization methods for engineering, Struct. Multidiscip. Optim. 26 (2004) 369–395.
[49] [Online], available: www.lindo.com.
[50] S. Gokhale, Architecture-based software reliability analysis: overview and limitations, IEEE Trans. Dependable Secure Comput. 4 (1) (2007) 32–40.
[51] J.M. Voas, K.W. Miller, Software testability: the new verification, IEEE Softw. 12 (1995) 17–28.
[52] D. Hamlet, D. Mason, D. Woit, Theory of software reliability based on components, in: Proceedings of the 23rd International Conference on Software Engineering, ICSE '01, 2001, pp. 361–370.
[53] R. Mirandola, P. Potena, P. Scandurra, An optimization process for adaptation space exploration of service-oriented applications: the stock trading system case study, Tech. rep., University of Bergamo, Italy, http://cs.unibg.it/potena/OptProcess/TRSTS.pdf.
[54] V. Cortellessa, F. Marinelli, P. Potena, Automated selection of software components based on cost/reliability tradeoff, in: Proc. of 3rd European Workshop on Software Architecture (EWSA 2006), in: Lecture Notes in Computer Science, vol. 4344, 2006, pp. 66–81.

[55] J. Voas, K. Miller, Software testability: the new verification, IEEE Softw. 12 (3) (1995) 17–28.
[56] K. Vallidevi, B. Chitra, Effective self adaptation by integrating adaptive framework with architectural patterns, in: Proc. of the 1st Amrita ACM-W Celebration on Women in Computing in India, A2CWiC '10, ACM, 2010.
[57] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture, vol. 1: A System of Patterns, Wiley, 1996.
[58] B.W. Boehm, Software Engineering Economics, 1st ed., Prentice Hall PTR, 1981.
[59] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[60] N.B. Harrison, P. Avgeriou, Incorporating fault tolerance tactics in software architecture patterns, in: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, SERENE '08, 2008, pp. 9–18.