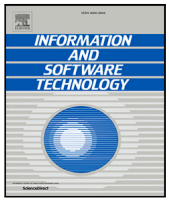
Information and Software Technology 157 (2023) 107159

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Many-objective optimization of non-functional attributes based on refactoring of software models

Vittorio Cortellessa [a,*,1], Daniele Di Pompeo [a,1], Vincenzo Stoico [a,1], Michele Tucci [b,1]

[a] University of L'Aquila, Italy
[b] Charles University, Czech Republic

ARTICLE INFO

Dataset link: https://github.com/SEALABQualityGroup/EASIER, https://github.com/SEALABQualityGroup/2022-ist-replication-package

Keywords:
Multi-objective optimization
Search-based software engineering
Performance
Reliability
Refactoring
Model-driven engineering
Software architecture

ABSTRACT

Context: Software quality estimation is a challenging and time-consuming activity, and models are crucial to face the complexity of such activity on modern software applications. In this context, software refactoring is a crucial activity within development life-cycles where requirements and functionalities rapidly evolve.
Objective: One main challenge is that the improvement of distinctive quality attributes may require contrasting refactoring actions on software, as for trade-off between performance and reliability (or other non-functional attributes). In such cases, multi-objective optimization can provide the designer with a wider view on these trade-offs and, consequently, can lead to identify suitable refactoring actions that take into account independent or even competing objectives.
Method: In this paper, we present an approach that exploits the *NSGA-II* as the genetic algorithm to search optimal Pareto frontiers for software refactoring while considering many objectives. We consider performance and reliability variations of a model alternative with respect to an initial model, the amount of performance antipatterns detected on the model alternative, and the architectural distance, which quantifies the effort to obtain a model alternative from the initial one.
Results: We applied our approach on two case studies: a Train Ticket Booking Service, and CoCoME. We observed that our approach is able to improve performance (by up to 42%) while preserving or even improving the reliability (by up to 32%) of generated model alternatives. We also observed that there exists an order of preference of refactoring actions among model alternatives.
Conclusion: Based on our analysis, we can state that performance antipatterns confirmed their ability to improve performance of a subject model in the context of many-objective optimization. In addition, the metric that we adopted for the architectural distance seems to be suitable for estimating the refactoring effort.

1. Introduction

Software refactoring [1] can be triggered by different causes, such as the introduction of additional requirements, the adaptation to new execution contexts, or the degradation of non-functional properties. The identification of optimal refactoring actions is a non-trivial task, mostly due to the large space of solutions, while there is still lack of automated support to this task. Search-based techniques have been involved in such a context [2–8], and they have proven to suit within the non-functional analysis due to the quantifiable nature of non-functional attributes [9–11]. Among the search-based techniques, those related to multi-objective optimization have been recently applied to model refactoring optimization problems [12,13]. A common aspect of multi-objective optimization approaches applied to model-based software refactoring problems is that they search among design alternatives (*e.g.,* through architectural tactics [13,14]).

In this paper, we present an approach based on a many-objective evolutionary algorithm (*i.e., NSGA-II* [15]) that searches sequences of refactoring actions, to be applied on models, leading to the optimization of four objectives: (i) performance variation (analyzed through Layered Queueing Networks [16]), (ii) reliability (analyzed through a closed-form model [17]), (iii) number of performance antipatterns (automatically detected [18]) and (iv) architectural distance [19]. A performance antipattern is a bad design decision that might lead to a performance degradation [20,21]. In particular, we analyze the composition of model alternatives generated through the application of refactoring actions to the initial model, and we analyze the contribution

* Corresponding author.
E-mail addresses: vittorio.cortellessa@univaq.it (V. Cortellessa), daniele.dipompeo@univaq.it (D. Di Pompeo), vincenzo.stoico@graduate.univaq.it (V. Stoico), tucci@d3s.mff.cuni.cz (M. Tucci).
[1] The contribution of each author is equivalent.

https://doi.org/10.1016/j.infsof.2023.107159
Received 1 March 2022; Received in revised form 4 October 2022; Accepted 20 January 2023
Available online 27 January 2023

of the architectural distance to the generation of Pareto frontiers. Furthermore, we study the impact of performance antipatterns on the quality of refactoring solutions. Since it has been shown that removing performance antipatterns leads to systems that show better performance than the ones affected by them [18,21,22], we aim at studying if this result persists in the context of many-objective optimization, where performance improvement is not the only objective.

Our approach applies to UML models augmented by MARTE [23] and DAM [24] profiles that allow to embed performance and reliability properties. However, UML does not provide native support for performance analysis, thus we introduce a model-to-model transformation that generates Layered Queueing Networks (LQN) from annotated UML models. The solution of LQN models feeds the performance variation objective.

Here, we consider refactoring actions that are designed to improve performance in most cases. Since such actions may also have an impact on other non-functional properties, we introduce the reliability among the optimization objectives to study whether satisfactory levels of performance and reliability can be kept at the same time. In order to quantify the reliability objective, we adopt an existing model for component-based software systems [17] that can be generated from UML models.

We also minimize the distance between the initial UML model and the ones resulting from applying refactoring actions. Indeed, without an objective that minimizes such distance, the proposed solutions could be impractical because they could require to completely disassemble and re-assemble the initial UML model.

In a recent work [25], we extended the approach in [12,19], by investigating UML models optimization, thus widening the scope of eligible models. In this paper, we extensively apply the approach to two case studies from the literature: Train Ticket Booking Service [26,27], and CoCoME [28]. We analyze the sensitivity of the search process to configuration variations. We refine the cost model of refactoring actions, introduced in [25], and we investigate how it contributes to the generation of Pareto frontiers. Also, we analyze the characteristics of computed Pareto frontiers in order to extract common properties for both case studies.

This study answers the following research questions:

- *RQ1*: To what extent do experimental configurations affect quality of Pareto frontiers?
  - *RQ1.1*: Does antipattern detection contribute to find better solutions compared to the case where antipatterns are not considered at all?
  - *RQ1.2*: Does the probabilistic nature of fuzzy antipatterns detection help to include higher quality solutions in Pareto frontiers with respect to deterministic one?
  - *RQ1.3*: To what extent does the architectural distance contribute to find better alternatives?
- *RQ2*: Is it possible to increase reliability without performance degradation?
- *RQ3*: What type of refactoring actions are more likely to lead to better solutions?

The experimentation lasted approximately *200* h and generated more than *70,000* model alternatives.

Generally, multi-objective optimization is beneficial when the solution space is so large that an exhaustive search is impractical. Hence, due to the search of the solution space, multi-objective optimization requires a lot of time and resources.

Our results show that, by considering the reduction of performance antipatterns as an objective, we are able to obtain model alternatives that show better performance and, in the majority of cases, better reliability as well. We also find that a more sophisticated architectural distance objective estimation helps the optimization process to generate model alternatives showing better quality indicators. Also, we strengthen the idea that performance antipatterns are promising proxies of performance degradation of software models.

The structure of the paper is the following: Section 2 introduces basic concepts, Section 3 describes the approach, Section 4 describes the two involved case studies, and Section 5 details used configurations, in Section 6 we evaluate our approach and discuss the results, threats to validity are described in Sections 7 and 8 reports related work, and Section 9 concludes the paper.

2. Background

We identify four competing objectives of our evolutionary approach as follows: *perfQ* is a performance quality indicator that quantifies the performance improvement/detriment between an initial model and one obtained by applying the refactoring actions of a solution (Section 2.1); *reliability* is a measure of the reliability of the software model (Section 2.2); *performance antipatterns* is a metric that quantifies the amount of performance antipattern occurrences while considering the intrinsic uncertainty rising from thresholds used by the detection mechanism (Section 2.3); *#changes* represents the distance between an initial model and one obtained by applying the refactoring actions of a solution (Section 2.4).

We employ the Non-dominated Sorting Algorithm II (*NSGA-II*) as our genetic algorithm [15], since it is extensively used in the software engineering community, *e.g.,* [14,29]. *NSGA-II* randomly creates an initial population of model alternatives, and it used to create the offspring population by applying the *Crossover* with probability $P_{crossover}$, and the *Mutation* with probability $P_{Mutation}$ operators. The union of the initial and the offspring populations is sorted by the *Non-dominated sorting* operator, which identifies different Pareto frontiers with respect to considered objectives. Finally, the *Crowding distance* operator cuts off the worse half of the sorted union population. Hence, the remaining model alternatives become the initial population for the next step.

2.1. Performance quality indicator (perfQ)

perfQ quantifies the performance improvement/detriment between two models, and it is defined as follows:

$$perfQ(M) = \frac{1}{c}\sum_{j=1}^{c} p_j \cdot \frac{F_j - I_j}{F_j + I_j}$$

where M is a model obtained by applying a refactoring solution to the initial model, F_j is the value of a performance index in M, and I_j is the value of the same index on the initial model. $p \in \{-1, 1\}$ is a multiplying factor that holds: (i) 1 if the jth index has to be maximized (i.e., the higher the value, the better the performance), like the throughput; (ii) -1 if the jth index has to be minimized (i.e., the smaller the value, the better the performance), like the response time.

Notice that, for performance measures representing utilization, p also holds 1 but we define a *utilization correction factor* Δ_j to be added to each jth term above, as defined in [19]. The utilization correction factor penalizes refactoring actions that push the utilization too close to 1, i.e., its maximum value. Finally, the global *perfQ* is computed as the average across the number c of performance indices considered in the performance analysis.

As mentioned in the introduction, in order to obtain performance indices of a UML model, the analysis has been conducted on Layered Queueing Networks (LQNs) [16][2] that are obtained through a model transformation approach from UML to LQN, which we have introduced in [25]. We chose Layered Queueing Networks as our performance model notation because it is extensively used in the literature and it allows a more explicit representation of software and hardware components (and their interactions) than the one of conventional Queueing Networks [13,14,30].

[2] http://www.sce.carleton.ca/rads/lqns/LQNSUserMan-jan13.pdf.

Table 1
Detectable performance antipatterns in our approach. Left column lists performance antipattern names, while right column lists performance antipattern descriptions [22].

Performance antipattern	Description
Pipe and filter	Occurs when the slowest filter in a "pipe and filter" causes the system to have unacceptable throughput.
Blob	Occurs when a single component either (i) performs the greatest part of the work of a software system or (ii) holds the greatest part of the data of the software system. Either manifestation results in excessive message traffic that may degrade performance.
Concurrent processing system	Occurs when processing cannot make use of available processors.
Extensive processing	Occurs when extensive processing in general impedes overall response time.
Empty semi-truck	Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both.
Tower of babel	Occurs when processes use different data formats and they spend too much time in convert them to an internal format.

2.2. Reliability model

The reliability model that we adopt here to quantify the *reliability* objective is based on the model introduced in [17]. The mean failure probability θ_S of a software system S is defined by the following equation:

$$\theta_S = 1 - \sum_{j=1}^{K} p_j \left(\prod_{i=1}^{N} (1-\theta_i)^{InvNr_{ij}} \cdot \prod_{l=1}^{L} (1-\psi_l)^{MsgSize(l,j)} \right)$$

This model takes into account failure probabilities of components (θ_i) and communication links (ψ_l), as well as the probability of a scenario to be executed (p_j). Such probabilities are combined to obtain the overall reliability on demand of the system (θ_S), which represents how often the system is not expected to fail when its scenarios are invoked.

The model is considered to be composed of N components and L communication links, whereas its behavior is made of K scenarios. The probability (p_j) of a scenario j to be executed is multiplied by an expression that describes the probability that no component or link fails during the execution of the scenario. This expression is composed of two terms: $\prod_{i=1}^{N}(1-\theta_i)^{InvNr_{ij}}$, which is the probability of the involved components not to fail raised to the power of their number of invocations in the scenario (denoted by $InvNr_{ij}$), and $\prod_{l=1}^{L}(1-\psi_l)^{MsgSize(l,j)}$, which is the probability of the involved links not to fail raised to the power of the size of messages traversing them in the scenario (denoted by $MsgSize(l,j)$).

2.3. Performance antipatterns

A performance antipattern describes bad design practices that might lead to performance degradation in a system. Smith and Williams have introduced the concepts of performance antipatterns in [21,31]. These textual descriptions were later translated into a first-order logic (FOL) equations [32].

A performance antipattern FOL is a combination of multiple literals, where each one represents a system aspect (*e.g.,* the number of connections among components). These literals must be compared to thresholds in order to reveal the occurrence of a performance antipattern. The identification of such thresholds is a non-trivial task, and using deterministic values may result in an excessively strict detection where the smallest change in the value of a literal determines the occurrence of the antipattern. For these reasons, we employ a fuzzy detection [33], which assigns to each performance antipattern a probability to be an antipattern. An example of a performance antipattern fuzzy detection is the following:

$$1 - \frac{UB(literal) - literal}{UB(literal) - LB(literal)}$$

The upper (UB) and the lower (LB) bounds, in the above equation, are the maximum and minimum values of the $literal$ computed on the entire system. Instead of detecting a performance antipattern in a deterministic way, such thresholds lead to assign probabilities to antipattern occurrences. In this study, we detect the performance antipatterns listed in Table 1.

2.4. Architectural distance

The architectural distance, that we express here as *#changes*, represents the distance of the model obtained by applying refactoring actions from the initial one [19]. On one side, a *baseline refactoring factor (BRF)* is associated to each refactoring action in our portfolio, and it expresses the refactoring effort to be spent when applying the action. On the other side, an *architectural weight (AW)* is associated to each model element on the basis of the number of connections to other elements in the model. Hence, we quantify the effort needed to perform a refactoring as the product between the *baseline refactoring factor* of an action and the *architectural weight* of the model element on which that action is applied. *#changes* is obtained by summing the efforts of all refactoring actions contained in a solution.

Furthermore, *BRF* and AW can assume any positive value (*i.e.,* zero is a non-admitted value because it would lead the optimizer to always select only actions by that type).

As an example, let us assume that a refactoring sequence is made up of two refactoring actions: A1 with $BRF(A1) = 1.23$, and A2 with $BRF(A2) = 2.3$. For each refactoring action, the algorithm randomly selects a target element in the model. For instance, let those target elements be: E1 with $AW(E1) = 1.43$, and E2 with $AW(E2) = 1.32$. The resulting *#changes* of A1 and A2 would be:

$$\#changes(A1, A2) = 1.23 \cdot 1.43 + 2.3 \cdot 1.32$$

Details about the *baseline refactoring factor* for each considered refactoring action are provided in Section 3.3.

3. Approach

Fig. 1 depicts the process we present in this paper. The process uses a UML model and a set of refactoring actions as input. The *Initial Model* and the *Refactoring Actions* are involved within the *Create Combined Population* step, where mating operations (*i.e.,* selection, mutation, and crossover) are put in place to create *Model Alternatives*. The mating operations randomly apply the refactoring actions, which generate alternatives functionally equivalent to the initial model. Therefore, the *Evaluation* step is applied to each model alternative. Subsequently, the model alternatives are ranked (*Sorting* step) according to four objectives: *perfQ*, *reliability*, *#changes*, and *performance antipatterns*. The optimal model alternatives (*i.e.,* non-dominated alternatives) become the input of the next iteration. The process continues until the stopping criteria are met. Finally, the process generates a *Pareto Frontier*, which contains all non-dominated model alternatives.

3.1. Assumptions on UML models

In our approach, we consider UML models including three views, namely *static, dynamic* and *deployment* views. The static view is modeled by a UML Component diagram in which static connections among components are represented by interface realizations and their usages. The dynamic view is described by UML Use Case and Sequence diagrams. A Use Case diagram defines user scenarios, while a Sequence

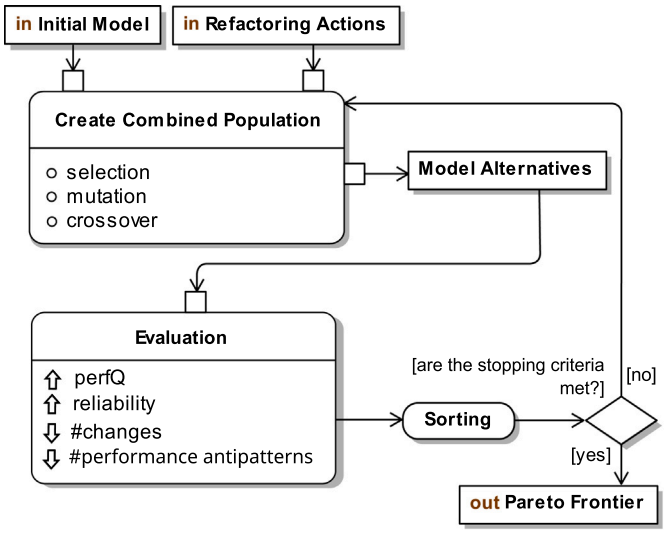


Fig. 1. Our multi-objective evolutionary approach.

diagram describes the behavior inside a single scenario through component operations (as defined in their interfaces) and interactions among them. A Deployment diagram is used to model platform information and map Components to Deployment Nodes. As mentioned before, we use an augmented UML notation by embedding two existing profiles, namely MARTE [23] that expresses performance concepts, and DAM [24] that expresses reliability concepts.

Although our assumptions on UML models seem to require an up-front modeling phase, the accuracy of results is affected by the quality of model and annotations. We mitigate the modeling effort through the usage of UML. In fact, a plethora of UML modeling tools is available, each equipped with entry-level or advanced capabilities that differently help software models design.[3]

3.2. The refactoring engine

The automated refactoring of UML models is a key point when evolutionary algorithms are employed in order to optimize some model attributes. For the sake of full automation of our approach, we have implemented a refactoring engine that applies refactoring actions on UML software models [34].

Each solution that our evolutionary algorithm produces is a sequence of refactoring actions that, once applied to an initial model, leads to a model alternative that shows different non-functional properties. Since our refactoring actions are combined during the evolutionary approach, we exploit the feasibility engine that verifies in advance whether a sequence of refactoring actions is feasible or not [35].

Our refactoring actions are equipped with pre- and post-condition. While the pre-condition represents the model state for enabling the action, the post-condition represents the model state when the action has been applied. The approach extracts a refactoring action and adds it to the sequence. As soon as the action is selected, it randomly extracts a model element (*i.e.,* the target element). Thus, the refactoring engine checks the feasibility of the (partial) sequence of refactoring actions. When the latest added action makes the sequence unfeasible, the engine discards the action and replaces it with a new one. The engine reduces a sequence of refactoring actions to a single refactoring action, which includes all the changes (see Eq. (1a)).

For example, considering two refactoring actions (M_i, and M_j), then the global pre-condition is obtained by logical ANDing the first action pre-condition (${}^{P_r}M_i$) and all the parts of M_j pre-condition that are not yet verified by M_i post-conditions ($M_j^{P_r} \;/\; M_i{}^{P_o}$) (see Eqs. (1b)). Since the status of the model after a refactoring is synthesized by its post-condition, we can discard the parts of a subsequent refactoring pre-condition that, by construction, are already verified by its post-condition. The global post-condition is obtained by logical ANDing all post-conditions within the sequence ($M_i^{P_o} \wedge M_j^{P_o}$) (see Eq. (1c)).

$$ {}^{P_r}M_i^{P_o} \wedge {}^{P_r}M_j^{P_o} \longmapsto {}^{P_r}M^{P_o} \tag{1a} $$

$$ {}^{P_r}M_i \wedge M_j^{P_r} \;/\; M_i{}^{P_o} \longmapsto {}^{P_r}M \tag{1b} $$

$$ M_i^{P_o} \wedge M_j^{P_o} \longmapsto M^{P_o} \tag{1c} $$

Our feasibility engine also allows to reduce the number of invalid refactoring sequences, thus reducing the computational time.

3.2.1. Refactoring action portfolio

Figs. 2 through 5 show a graphic representation of each refactoring action. Each figure's left side shows the original model (*e.g.,* static view in Fig. 3(a), dynamic view in Fig. 3(c), and deployment view in Fig. 3(e)), while the refactored version is shown on the right side (*e.g.,* static view in Fig. 3(b), dynamic view in Fig. 3(d), and deployment view in Fig. 3(f)). The red highlights indicate changes.

Clone a node (Clon). This action is aimed at introducing a replica of a Node. Adding a replica means that every deployed artifact and every connection of the original Node has to be in turn cloned. Stereotypes and their tagged values are cloned as well. The rationale of this action is to introduce a replica of a platform device with the aim of reducing its utilization (see Fig. 2).

Move an operation to a new component deployed on a new node (MO2N). This action is in charge of randomly selecting an operation and moving it to a new Component. All the elements related to the moving operation (*e.g.,* links) will move as well. Since we adopt a multi-view model, and coherence among views has to be preserved, this action has to synchronize dynamic and deployment views. A lifeline for the newly created Component is added in the dynamic view, and messages related to the moved operation are forwarded to it. In the deployment view, instead, a new Node, a new artifact, and related links are created. The rationale of this action is to lighten the load of the original Component and Node (see Fig. 3).

Move an operation to a component (MO2C). This action is in charge of randomly selecting and transferring an Operation to an arbitrary existing target Component. The action consequently modifies each UML Use Case in which the Operation is involved. Sequence Diagrams are also updated to include a new lifeline representing the Component owning the Operation, but also to re-assign the messages invoking the operation to the newly created lifeline. The rationale of this action is quite similar to the previous refactoring action, but without adding a new UML Node to the model (see Fig. 4).

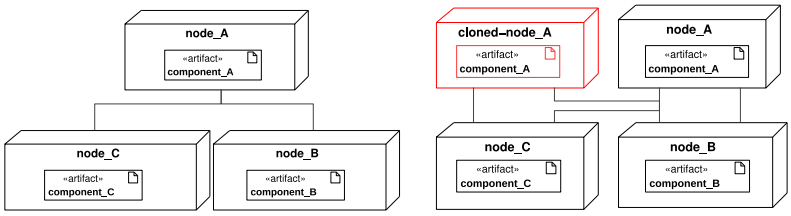
Deploy a component on a new node (ReDe). This action simply modifies the deployment view by redeploying a Component to a newly created Node. In order to be consistent with the initial model, the new Node is connected with all other ones directly connected to the Node on which the target Component was originally deployed. The rationale of this action is to lighten the load of the original UML Node by transferring the load of the moving Component to a new UML Node (see Fig. 5).

3.3. Baseline refactoring factor

As described in Section 2.4, we measure the architectural distance by summing the products of baseline refactoring factor (*BRF*) and architectural weight (AW) for each refactoring action $a_i(el_j)$ within a sequence ($\mathbb{A}$).

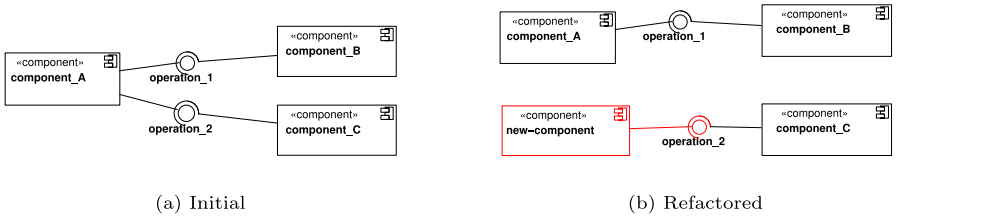
$$ \#changes(\mathbb{A}) = \sum_{a_i(el_j)\in\mathbb{A}} BRF(a_i) \times AW(el_j) $$

[3] https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools.

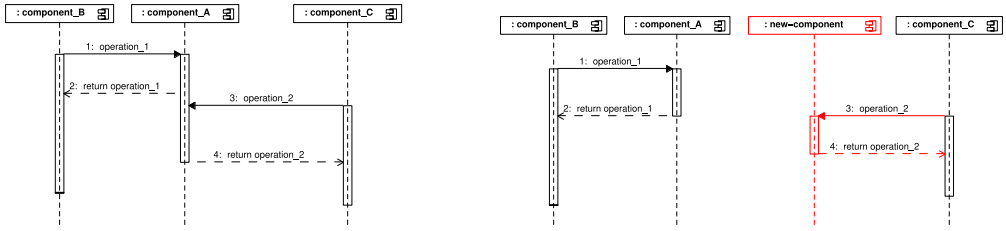


(a) Initial (b) Refactored

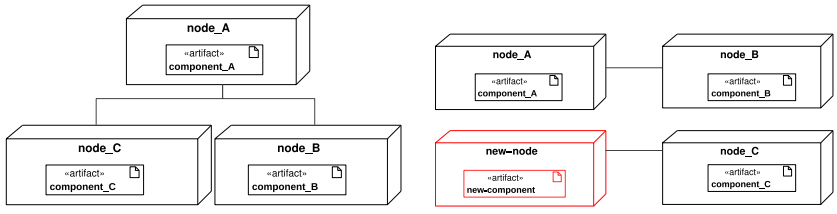
Fig. 2. The *Clon* refactoring action example on *node_A* through a UML software model.



(a) Initial (b) Refactored

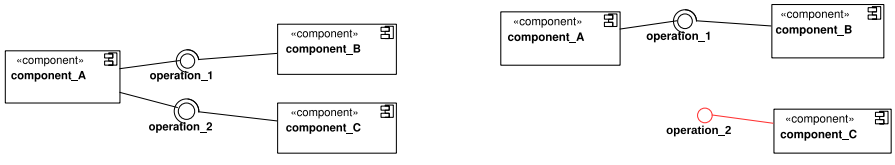


(c) Initial (d) Refactored

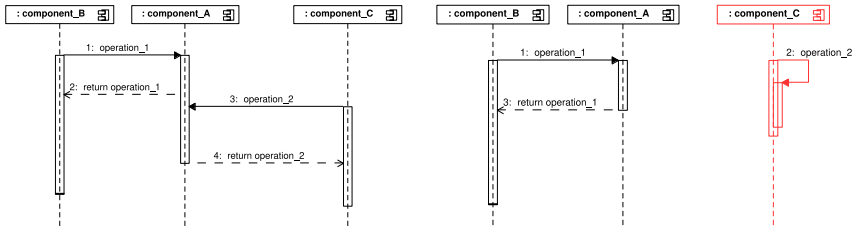


(e) Initial (f) Refactored

Fig. 3. The *MO2N* refactoring action example on *operation_2* through a UML software model.

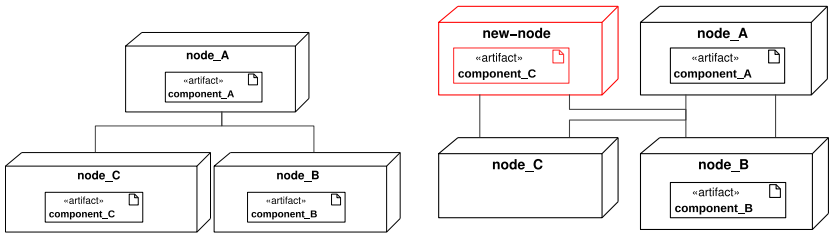


(a) Initial (b) Refactored



(c) Initial (d) Refactored

Fig. 4. The *MO2C* refactoring action example on *operation_2* and *component_C* through a UML software model.



(a) Initial (b) Refactored

Fig. 5. The *ReDe* refactoring action example on *component_C* through a UML software model.

Table 2
A detailed size of the solution space (Ω) computation.

Action	*BRF*	TTBS	CoCoME
MO2N	1.80	70	$\approx 4.8 \times 10^3$
MO2C	1.64	$\approx 1.5 \times 10^6$	$\approx 1.3 \times 10^8$
ReDe	1.45	$\approx 3 \times 10^2$	$\approx 7 \times 10^2$
Clon	1.23	$\approx 3 \times 10^2$	70
Ω		9.45×10^{12}	3.05×10^{16}

AW is the weight of the target of the refactoring action, *BRF* is the intrinsic cost that one should pay in order to apply the specific action on a model element. There are different ways to compute the effort for implementing software artifacts or maintaining them (*e.g.,* COCOMO-II [36], and CoBRA [37]). Nevertheless, we consider the cost in terms of the effort that one should spend on the model to complete a refactoring action, and we assign *BRF* values on the basis of our past experience in manual refactoring. We have not used a cost estimator model, such as CoBRA, because it requires to collect business information that is not available for non-industrial case studies. Table 2 lists the *BRF* values used in this study. It is worth remarking that, in our optimization problem, the ratio among *BRF* values is more important than how each single value has been extracted.

3.4. Computing reliability on UML models

The reliability parameters of the model introduced in Section 2.2 are annotated on UML models by means of the MARTE-DAM profile. The probability of executing a scenario (p_j) is specified by annotating UML Use Cases with the *GaScenario* stereotype. This stereotype has a tag named *root* that is a reference to the first *GaStep* in a sequence. We use the *GaScenario.root* tag to point to the triggering UML Message of a Sequence Diagram and the *GaStep.prob* to set the execution probability. Failure probabilities of components (θ_i) are defined by applying the *DaComponent* stereotype on each UML Component and by setting, in the *failure* tag, a *DaFailure* element with the failure probability specified in the *occurrenceProb* tag. Analogously, failure probabilities of links (ψ_l) are defined in the *failure.occurrenceProb* tag of the *DaConnector* stereotype that we apply on UML CommunicationPath elements. Such elements represent the connection links between UML Nodes in a Deployment Diagram. Sequence Diagrams are traversed to obtain the number of invocations of a component i in a scenario j (denoted by $InvNr_{ij}$ in our reliability model), but also to compute the total size of messages passing over a link l in a scenario j (denoted by $MsgSize(l, j)$). The size of a single UML Message is annotated using the *GaStep.msgSize* tag. The Java implementation of the reliability model is available online.[4]

[4] https://github.com/SEALABQualityGroup/uml-reliability.

3.5. Pareto frontier quality indicators

We compare the performance of the *NSGA-II* while varying the configuration eligible values listed in Table 3. We used well-established quality indicators also provided in the JMetal framework [38]. We use quality indicators to quantify the difference among computed Pareto frontiers (PF^c) with respect to the reference Pareto frontier (PF^{ref}) [39]. Therefore, we can declare which configuration outperform the others.

In the following, we recall some characteristics for each quality indicator.

GSPREAD. The Generalized SPREAD is a quality indicator to be minimized, and it measures the spread of solution within PF^c [40]. It is computed as follows:

$$GSPREAD(PF^c) = \frac{\sum_{i=1}^{m} d(e_i, PF^c) + \sum_{s \in PF^c} |id(s, PF^c) - \bar{id}|}{\sum_{i=0}^{m} d(e_i, PF^c) + |PF^c| * \bar{id}}$$

where e_i is the optimal value for the objective f_i, *i.e.,* $(e_1, \ldots, e_m)$ is the extreme solution in PF^{ref}, $id(s, PF^c) = d(s, PF^c \backslash \{s\})$ is the minimal distance of a solution s from the solutions in PF^c, and $\bar{id}$ is the mean value of $id(s, PF^c)$ across the solutions s in PF^{ref}.

IGD$^+$. The Inverse Generational Distance plus is a quality indicator to be minimized. It measures the distance from a solution in PF^{ref} to the nearest solutions in PF^c [41]. It is computed as follows:

$$IGD^+(PF^c) = \frac{\sqrt{\sum_{s \in PF^{ref}} d(s, PF^c)^2}}{|PF^{ref}|}$$

Hypervolume. The Hypervolume indicator is to be maximized and it measures the volume of the solution space Ω covered by PF^c [42]. It is computed as follows:

$$HV(PF^c) = volume(\cup_{s_i \in PF^c} hc(s_i))$$

where s_i is a solution within the PF^c, $hc(s_i)$ is the hypercube having s_i and w as diagonal points. The variable w is the reference point computed using the worst objective function values among all the possible solutions in PF^c.

EPSILON. The EPSILON quality indicator measures the smallest distance that each solution within PF^c should be translated so that PF^c dominates PF^{ref} [43]. EPSILON is a quality indicator to be minimized, and it uses the notation of epsilon-dominance $\succ_\epsilon$. It is computed as follows:

$$EP(PF^c) = inf\{\epsilon \in \mathbb{R} | (\forall x \in PF^{ref}, \exists y \in PF^c : y \succ_\epsilon x)\}$$

In our study, we have computed a PF^{ref} for each case study by extracting every non-dominated solutions across each PF^c, *i.e.,* one for each configuration. Hence, the quality indicators in Tables 5 and 6 have been computed with respect to the PF^{ref} for the TTBS, and CoCoME case study respectively.

Table 3
Eligible configuration values.

	Configuration	Eligible values
Experiment settings	Baseline refactoring factor	No, yes
	Performance antipattern fuzziness	0.55, 0.80, 0.95
	Case study	TTBS, CoCoME
NSGA-II	Number of genetic evolutions	72, 82, 102
	Population size	16
	Number of independent runs	3
	Selection operator	Binary tournament selection
	$P_{crossover}$	0.80
	Crossover operator	Single point
	$P_{mutation}$	0.20
	Mutation operator	Simple mutation

4. Case studies

In this section, we apply our approach to the Train Ticket Booking Service (TTBS) case study [26,27], and to the well-established model case study CoCoME, whose UML model has been derived by the specification in [28].

4.1. Train Ticket Booking Service

Train Ticket Booking Service (TTBS) is a web-based booking application, whose architecture is based on the microservice paradigm. The system is made up of 40 microservices, and it provides different scenarios through users that can perform realistic operations, *e.g.,* book a ticket or watch trip information like intermediate stops. The application employs a docker container for each microservice, and connections among them are managed by a central pivot container.

Our UML model of TTBS is available online.[5] The static view is made of **11** UML Components, where each component represents a microservice. In the deployment view, we consider **11** UML Nodes, each one representing a docker container.

Among all TTBS scenarios shown in [26], in this paper we have considered **3** UML Use Cases, namely *login*, *update user details* and *rebook*. We selected these three scenarios because they commonly represent performance-critical ones in a ticketing booking service. Each scenario is described by a UML Sequence Diagram. Furthermore, the model comprises two user categories: simple and admin users. The simple user category can perform the login and the rebook scenarios, while the admin category can perform the login and the update user details scenarios.

4.2. CoCoME

The component-based system engineering domain has always been characterized by a plethora of standards for implementing, documenting, and deploying components. These standards are well-known as component models. Before the birth of the common component modeling example (CoCoME) [28], it was hard for researchers to compare different component models. CoCoME is a case study that acts as a single specification to be implemented using different component models.

CoCoME describes a Trading System containing several stores. A store might have one or more cash desks for processing goodies. A cash desk is equipped with all the tools needed to serve a customer (e.g., a Cash Box, Printer, Bar Code Scanner). CoCoME covers possible scenarios performed at a cash desk (e.g., scanning products, paying by credit card, generating reports, or ordering new goodies). A set of cash desks forms a cash desk line. The latter is connected to the store server for registering cash desk line activities. Instead, a set of stores is organized in an enterprise having its server for monitoring stores operations.

CoCoME describes 8 scenarios involving more than 20 components. We have modeled this case study using UML and following the structure described in Section 3.1. From the CoCoME original specification, we analyzed different operational profiles, *i.e.,* scenarios triggered by different actors (such as Customer, Cashier, StoreManager, StockManager), and we excluded those related to marginal parts of the system, such as scenarios of the *EnterpriseManager* actor. Thus, we selected **3** UML Use Cases, **13** UML Components, and **8** UML Nodes from the CoCoME specification. Beside this, we focused on three scenarios, namely: UC1 that describes the arrival of a customer at the checkout, identification, and sale of a product; UC4 that represents how products are registered in the store database upon their arrival; UC5 that represents the possibility of generating a report of store activities.

We computed the size of the solution space (Ω) as the Cartesian product of the combination of refactoring actions $C_{n,k} = \binom{n}{k}$ where n is the number of target model elements, and k is the length of the chromosome (*i.e.,* the length of the sequence of refactoring actions, which is 4 in our case), and we summarize data in Table 2. We remark that a manual investigation of the solution space is unfeasible due to its size. Hence, the evolutionary search is helpful for looking for model alternatives showing better quality than the initial one. Table 4 summarizes the case study characteristics.

Table 4
Number of UML elements in our Case Studies, and the size of the relative solution space (Ω).

Case study	UML node	UML component	UML message	Ω
TTBS	11	11	8	1.20×10^{13}
CoCoME	8	13	20	3.26×10^{16}

5. Experimental setup

A configuration is defined by the combination of parameters related to the genetic algorithm, and the ones related to the specific optimization model. The eligible configuration values in our approach are listed in Table 3. In order to investigate which configuration produces better Pareto frontiers, we have executed multiple tuning runs to find a set of optimal configurations.

In order to set the parameters related to the genetic algorithm, we have performed a tuning phase with the intent of increasing the quality of the Pareto frontiers. In particular, we have set the length of refactoring sequences to four actions, which represents a good approximation of the number of refactoring actions usually applied by a designer in a single session. We have set the $P_{crossover}$ and $P_{mutation}$ probabilities to 0.8 and 0.2, respectively, following common configurations [44]. The higher the values of these two probabilities, the greater the chance of generating an unfeasible sequence of refactoring actions, which in turn causes a longer simulation time due to a higher number of discarded sequences. For example, the $P_{crossover}$ increase could cause a lot of

[5] https://github.com/SEALABQualityGroup/2022-ist-replication-package/tree/main/case-studies/train-ticket.

permutation among sequences, and it might lead to wrong or unfeasible sequences of refactoring actions.

The initial population size might drive the genetic algorithm in local minima, and thus result in stagnant solutions. In general, a densely populated initial population minimizes the probability of stagnant solutions in local minima. However, the generation of a crowded initial population is computational demanding and, in case of rare local minima, the computational cost represents a clear slowdown for the evolutionary approach [45]. For that reason, we set the population size to **16** elements (i.e., 16 different UML model alternatives), which did not show stagnant issues in our tuning phase. Furthermore, we will investigate in a future work the impact of denser populations in our analysis, in terms of computational time and quality of the computed Pareto frontiers (PF^c). In addition, multiple runs have been executed for each configuration in order to reduce the randomness of the genetic algorithm.

We considered three fuzziness thresholds, *i.e.,* {0.55, 0.80, 0.95}, to study the impact of performance antipatterns on computed Pareto frontiers. Since we are considering a fuzzy detection of performance antipatterns, we should use values greater than 50% to reduce the probability of false positives, but less than 100% to not fall in a case of performance antipatterns deterministic detection. Therefore, we decided to use those three fuzziness values to analyze the uncertainty of a fuzzy performance antipatterns detection.

With regard to parameters related to refactoring actions, we ran the experiment twice, one by excluding *BRF*, and one by including it. For the latter, we set *BRF* of each refactoring action as reported in Table 2. As we said in Section 3.3, we did not employ a complex cost model for baseline refactoring factor values. However, we remark that we are interested in the ratio between *BRF* values rather than in their specific values, and we will deeply investigate the impact of other values on future work.

Our experimental settings on TTBS and CoCoME case studies have generated *70,000* model alternatives and have taken *200* h of computation. We performed our experiments on a server equipped with two Intel Xeon E5-2650 v3 CPUs at 2.30 GHz, 40 cores and 80 GB of RAM.

6. Results and discussion

Results presented in this section are aimed at answering the aforementioned three research questions.

6.1. RQ1

RQ1: To what extent do experimental configurations affect quality of Pareto frontiers?

RQ1 focuses on the contribution of experimental configurations to the quality of the computed Pareto frontiers (PF^c).

In Tables 5 and 6 it is possible to observe the configurations that result in better Pareto frontiers. Generally, quality indicators are obtained with respect to the optimal reference Pareto frontier (PF^{ref}), and each one has its ideal value (*e.g.,* $HV = 1$, $IDG^+ = 0$). Moreover, values in tables have been sorted in ascending order when the best quality indicator is the lowest one, and in descending order otherwise. Since we did not have the optimal PF^{ref} for our case studies, we computed, for each case study, the quality indicators with respect to a PF^{ref} that contains every non-dominated solution across all PF^c. Once quality indicators have been obtained and sorted, we identify which *maxeval* and *probpas* have generated better indicators. Finally, we also report data about *BRF*.

At a glance, we can see that in most cases for both case studies, $maxeval = 72$ and lower fuzziness generates better quality indicators, whereas *BRF* has a different impact on the two case studies.

Table 5
Best five of each quality indicator for the Train Ticket Booking Service case study while varying the performance antipattern fuzziness and the genetic algorithm evolutions.

BRF	*maxeval*	*probpas*	q_indicator	Value
yes	72	95	HV	0.329645
yes	82	95	HV	0.304931
yes	82	95	HV	0.267898
yes	72	80	HV	0.266588
yes	82	55	HV	0.254973
yes	72	95	IGD^+	0.135226
yes	82	95	IGD^+	0.149903
yes	72	55	IGD^+	0.157150
yes	82	95	IGD^+	0.167142
yes	72	80	IGD^+	0.173162
yes	72	95	EP	0.295681
yes	82	95	EP	0.296014
yes	82	95	EP	0.316964
yes	72	55	EP	0.316964
yes	82	55	EP	0.323661
yes	102	55	GSPREAD	0.125487
yes	102	95	GSPREAD	0.127085
yes	102	80	GSPREAD	0.144666
yes	102	55	GSPREAD	0.148802
yes	72	55	GSPREAD	0.203504

Table 6
Best five of each quality indicator for the CoCoME case study while varying the performance antipattern fuzziness and the genetic algorithm evolutions.

BRF	*maxeval*	*probpas*	q_indicator	Value
no	72	95	HV	0.360432
no	82	95	HV	0.359415
no	102	95	HV	0.342563
no	72	55	HV	0.326384
no	82	95	HV	0.305201
no	72	95	IGD+	0.091767
no	82	95	IGD+	0.105173
no	102	95	IGD+	0.106406
no	82	95	IGD+	0.132800
no	72	55	IGD+	0.135904
no	82	95	EP	0.250000
no	72	55	EP	0.250000
no	72	95	EP	0.250000
no	82	95	EP	0.313857
yes	72	95	EP	0.333333
no	82	55	GSPREAD	0.145989
yes	102	55	GSPREAD	0.193488
yes	102	95	GSPREAD	0.196790
no	102	55	GSPREAD	0.200320
no	102	80	GSPREAD	0.203431

In the following, we split *RQ1* into three sub-questions, each one related to a specific experimental configuration attribute. *RQ1.1* analyzes the influence of performance antipatterns on PF^c. *RQ1.2* investigates whether the fuzziness of performance antipattern detection helps to find better PF^c. *RQ1.3* studies the contribution of *BRF* to the quality of PF^c.

6.1.1. RQ1.1

RQ1.1: Does antipattern detection contribute to find better solutions compared to the case where antipatterns are not considered at all?

In order to answer this research question, we have conducted an additional experimentation for every problem configuration, where we have removed performance antipattern occurrences from the fitness function, thus reducing the optimization to the remaining three objectives.

Train Ticket Booking Service. Fig. 6 depicts the Pareto frontiers of 72 genetic evolutions while considering the lowest fuzziness (*i.e.,*

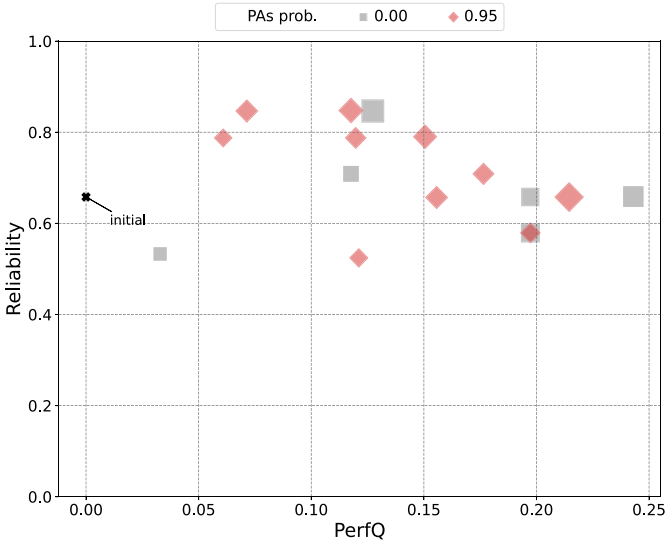


Fig. 6. The scatter plot of PF^c of TTBS with 72 genetic evolutions while considering, and excluding performance antipatterns in the optimization process (*i.e.,* $probpas = 0.95$, and $probpas = 0$).

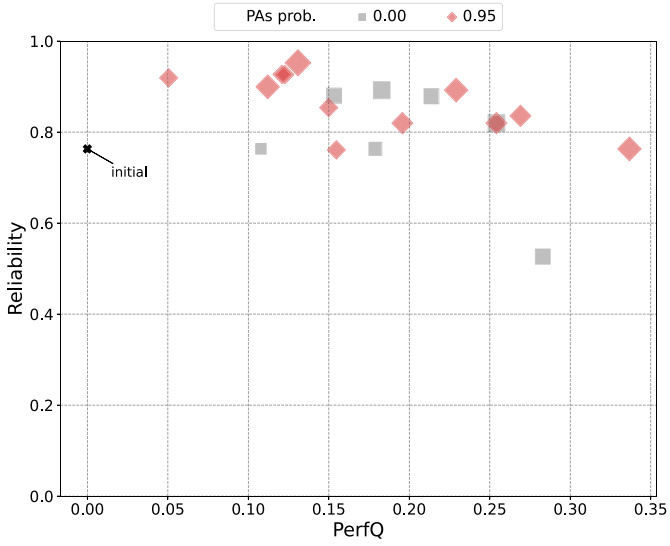


Fig. 7. The scatter plot of PF^c of CoCoME 72 genetic evolutions while considering, and excluding performance antipatterns in the optimization process (*i.e.,* $probpas = 0.95$, and $probpas = 0$).

$probpas = 0.95$) and no performance antipatterns (*i.e.,* $probpas = 0$). We can see that frontiers with performance antipatterns are generally more densely populated than the case where $probpas = 0$. Also, performance antipatterns help finding model alternatives showing lower *#changes* than the ones found when they have been ignored. Although $probpas = 0$ generates the highest value of *perfQ* (*i.e.,* $perfQ = 0.24$), there are more solutions in the topmost part of the plot when performance antipatterns drive the search process. From our analysis, it emerges that $probpas = 0.95$ produces better frontiers among those with performance antipatterns. Therefore, we can state that, for the TTBS case study, the lower fuzziness the better the quality of frontiers in terms of *perfQ*, *reliability*, and *#changes*.

CoCoME. Fig. 7 depicts the Pareto frontiers with 72 genetic evolutions while considering the lowest fuzziness (*i.e.,* $probpas = 0.95$) and no performance antipatterns (*i.e.,* $probpas = 0$). Most of the solutions lay in the topmost part of the plot, thus meaning that PF^c shows better *perfQ* and *reliability* of the initial solution (see the black cross in the figure). Frontiers generated by performance antipatterns are more densely populated than those without performance antipatterns. Thus, the reduction of the number of performance antipatterns occurrences, if it is included among the objectives, helps the process finding more alternative models showing higher *perfQ* and *reliability* with lower *#changes*.

Discussion. Based on our analysis, the reduction of performance antipatterns helps the optimization problem to generate alternatives showing better performance and reliability in most of the cases. The CoCoME case study has mainly shown a light search for better reliability, likely due to the high reliability value of the initial model.

On the basis of our experimentation, we can state that the consideration of performance antipattern occurrences in the optimization process leads to better solutions than the ones found when ignoring them.

6.1.2. RQ1.2

RQ1.2: Does the probabilistic nature of fuzzy antipatterns detection help to include higher quality solutions in Pareto frontiers with respect to the deterministic one?

In order to answer this research question, we varied the values of the fuzziness threshold of the performance antipatterns detection within {0.50, 0.80, 0.95} for the two case studies. Figs. 8 and 9 depict the kernel density estimate (KDE) plots showing each possible combination among objectives for TTBS and CoCoME respectively. Each plot depicts the KDE of the relative objectives, *e.g.,* Fig. 8(a) shows the *perfQ* KDE for the TTBS case study.

Train Ticket Booking Service. For the TTBS case study, we have noticed larger variability of *perfQ* when performance antipatterns are ignored, see the flattest curve in Fig. 8(a). In addition, *perfQ* is narrower to the mean (≈0.2) when performance antipatterns are involved in the fitness function, which means less variability in terms of performance in the model alternatives. With regard to the *reliability* (Fig. 8(b)), it seems to be more stable without performance antipattern detection. Moreover, the performance antipattern detection helps including solutions with higher reliability values than the case without them. Fig. 8(c) shows that the lower the fuzziness the more stable the *#changes* values, which means less variability in the model alternatives discovered by the search. Finally, the 0.95 fuzziness reduces the variability of the *performance antipatterns* objective (Fig. 8(d)). Thus, the more deterministic, the higher the probability of discovering true positive performance antipatterns.

CoCoME. We notice that Pareto frontiers obtained while ignoring performance antipatterns in the fitness function showed larger variability than the ones obtained while considering them. This is depicted in Fig. 9(a) where *perfQ* shows negative values and the curve is flatter than the other cases. For CoCoME we notice that the higher the performance antipattern *probpas*, the higher *perfQ*, which becomes similar to a normal distribution with mean falling on 0.3 for a $probpas = 0.95$ of performance fuzziness. In the case of the lowest fuzziness value, *perfQ* assumed the highest value in our experiments. With regard to *#changes* (Fig. 9(c)), it increases when *performance antipatterns* are ignored. Moreover, the higher the *probpas*, the more stable *#changes*, which means less variability in the model alternatives. Again, due to the high value of *reliability* for the initial model, CoCoME shows most of the *reliability* values around 0.9 (Fig. 9(b)).

Discussion. Our analysis shows that in most of the cases the higher *probpas*, the closer to the mean is the distribution of *perfQ*, which means less variability for *perfQ*. Therefore, it seems better to use a

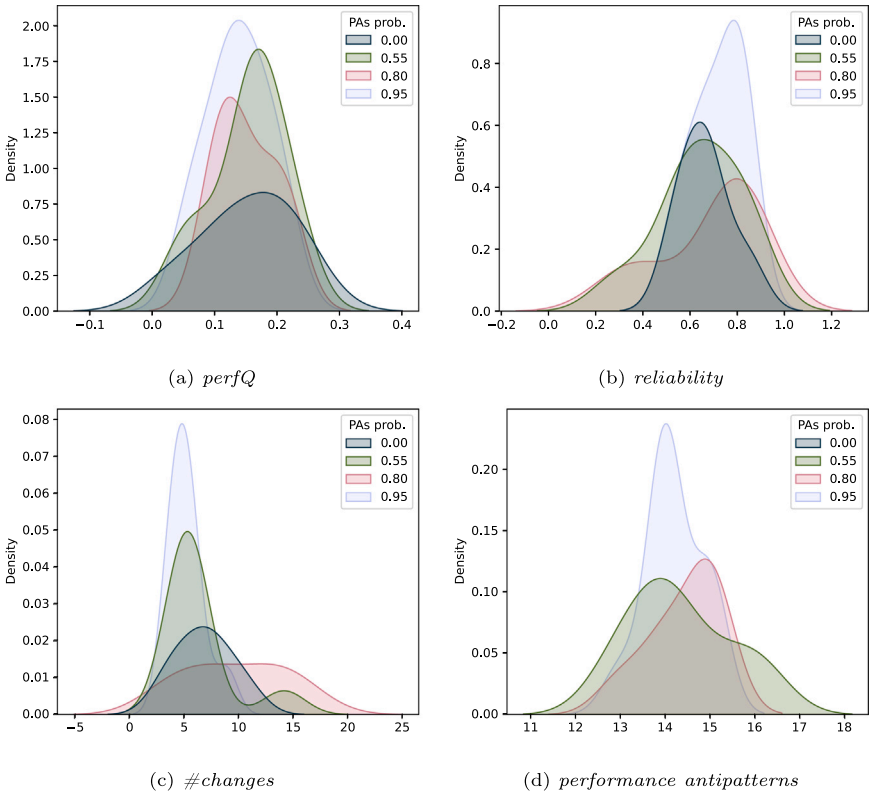


Fig. 8. The KDE plots of the Train Ticket Booking Service case study while varying the performance antipattern fuzziness probabilities. The *probpas* = 0.00 means performance antipatterns were ignored as objectives. Each plot is referring to the objective in the label.

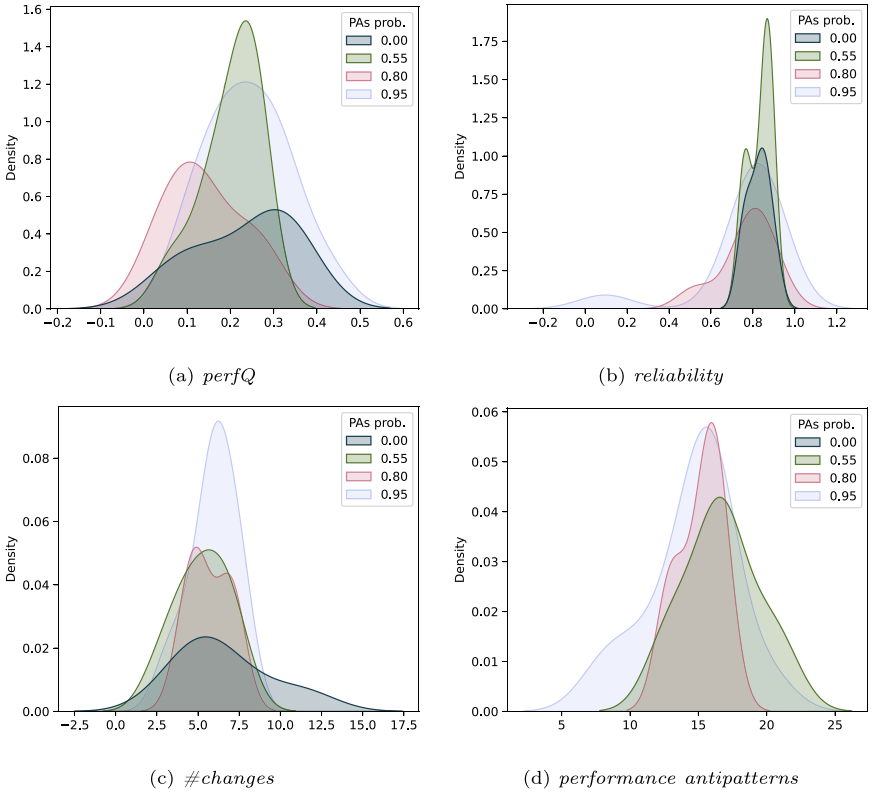


Fig. 9. The KDE plots of the CoCoME case study while varying the performance antipattern fuzziness probabilities. The *probpas* = 0.00 means performance antipatterns were ignored as objectives. Each plot is referring to the objective in the label.

more deterministic antipattern detection (*i.e.,* higher values of *probpas*). However, a deterministic detection has the drawback of relying on fixed thresholds that must be computed in advance for each model alternative. The trade-off between better quality solution and the effort to bind thresholds is likely domain-dependent and worth to be more investigated.

On the basis of our experimentation, we can state that performance antipattern fuzzy detection does not help to improve the quality of Pareto frontiers.

6.1.3. RQ1.3

RQ1.3: To what extent does the architectural distance contribute to find better alternatives?

In order to answer this research question, we run the same problem configurations by varying the baseline refactoring factor value. In particular, we decided to activate (*BRF*) and deactivate (*noBRF*) the baseline refactoring factor to study how it contributes to the generation of Pareto frontiers.

Train Ticket Booking Service. Figs. 10(a) and 10(b) show Pareto frontiers obtained with *BRF* and *noBRF* configurations, respectively. We can see that results with *noBRF* are narrower to the initial solution (*i.e.,* the black marker in figure) than the case where *BRF* is activated. *noBRF* seems to penalize performance antipatterns with higher fuzziness, in fact $probpas = 0.95$ generates the best alternatives in terms of *perfQ* and *reliability* (see the topmost right corner in Fig. 10(b)). However, the highest *perfQ* in the case of *noBRF* is lower than the one in the case of *BRF*. Hence, *BRF* helps the search finding better solutions in terms of *perfQ* for the TTBS case. Also, the *noBRF* configuration shows, in a few cases, a detriment of the initial performance and reliability (see the left bottom-most corner) that it never happened when the *BRF* is active.

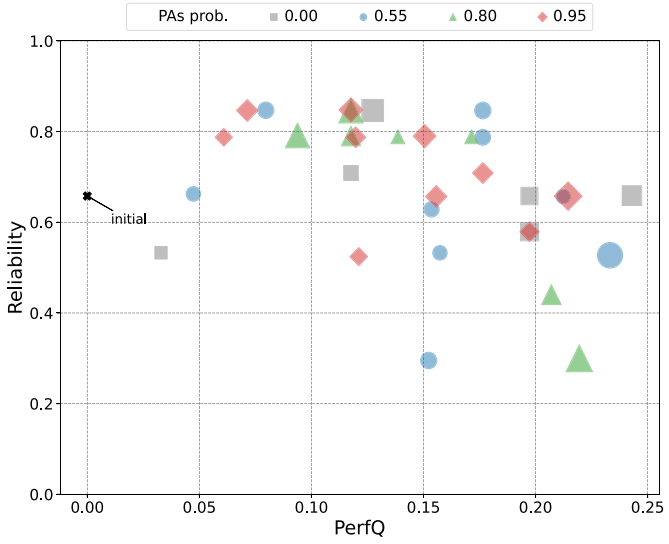
CoCoME. Fig. 11(b) shows Pareto frontiers obtained with *noBRF* configuration. By comparing this plot with the one shown in Fig. 11(a), we can see that the *BRF* exclusion generates more densely populated frontiers than the other case. Furthermore, no extreme differences arise between the executions with *BRF* and *noBRF* configurations. In both cases *perfQ* and *reliability* fall within the same region of the plot, where alternatives with *BRF* reached better *perfQ* (see $perfQ > 0.4$ in Fig. 11(a)). With regard to the *reliability*, we can see that *noBRF* configuration found few model alternatives showing lower values.

Discussion. Based on our analysis, the *baseline refactoring factor* helps generating better alternatives in terms of objectives. We noticed that the *reliability* is penalized with *noBRF* configurations. Also, the *BRF* deactivation penalized *perfQ* in few cases. A deeper investigation is required on how *BRF* might affect the computed Pareto frontiers quality. For example, we can introduce more complex cost models, *e.g.,* COCOMO [36], to improve its estimation. However, we preferred having a more straightforward cost estimation to avoid burdening the search algorithm with additional computational costs.

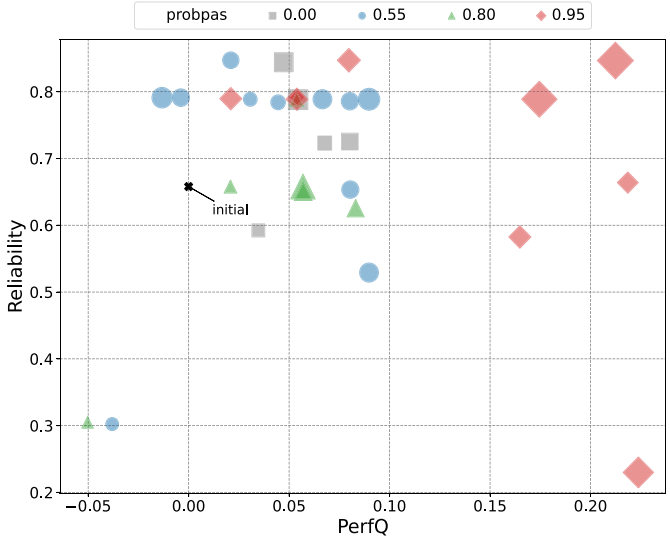
Based on our results, we can state that BRF helps better estimating #changes of refactoring actions, which generates Pareto frontiers showing higher quality (or at least it does not worsen the Pareto frontier quality).

6.2. RQ2

RQ2: Is it possible to increase reliability without performance degradation?



(a) BRF.



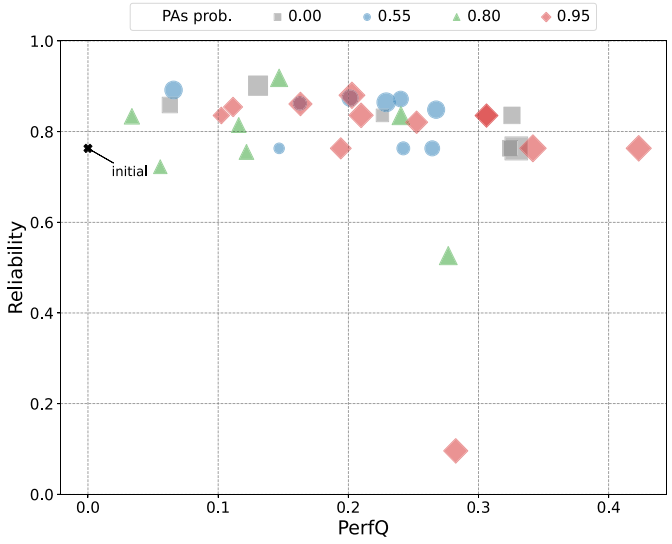
(b) $noBRF$.

Fig. 10. The scatter plot of Train Ticket Booking Service Pareto frontiers while varying the fuzziness after 72 genetic evolutions with *BRF*, and *noBRF* configurations.

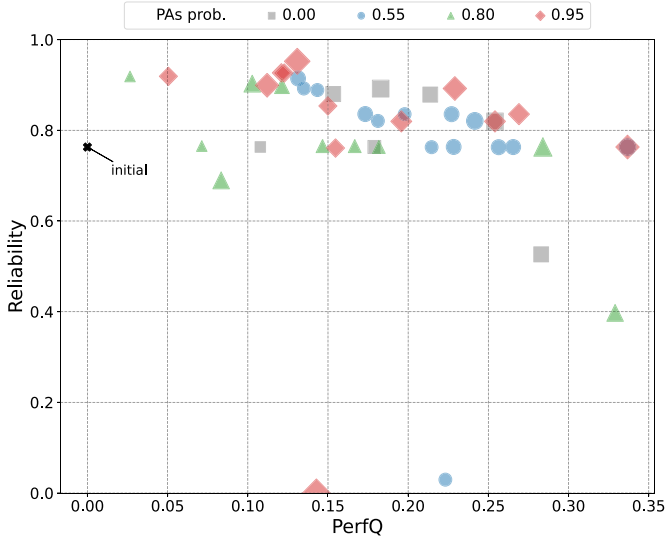
We answer RQ2 by looking for model alternatives, within the computed Pareto frontiers (PF^c), that improve both initial reliability and performance.

Fig. 12 shows the results obtained on the PF^c of TTBS and CoCoME. The dark dots represent the alternatives we are looking for, *i.e.,* those improving both *reliability* and *perfQ*. Instead, the bright dots represent the model alternatives that improve one of the two non-functional aspects.

Train Ticket Booking Service. Fig. 12(a) shows that in TTBS we obtained 54% of the model alternatives improving *reliability* and *perfQ*. Thus, there is a portion (*i.e.,* 46%) presenting a detriment of the *reliability* but an improvement in terms of performance. This is confirmed by looking at the model alternatives within the PF^{ref}: 18 over 26 alternatives are those taken from the examined Pareto. In this case, model alternatives that guarantee an improvement can be very important for a designer, as we find a performance upgrade of up to 27% and a reliability increase of up to 32%.

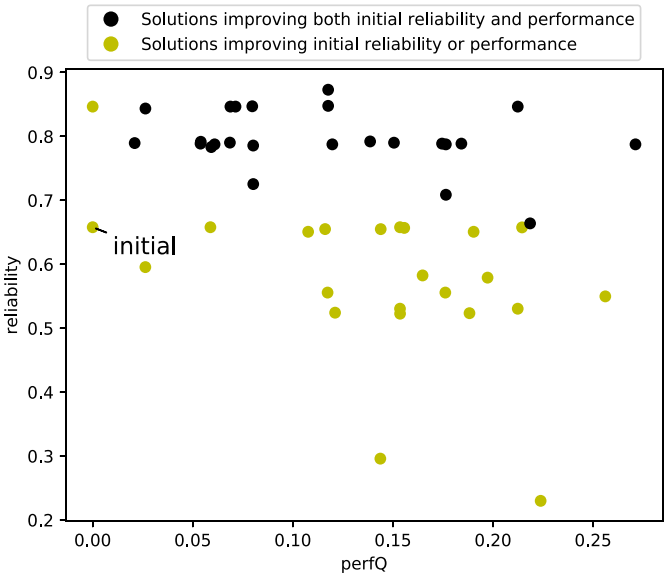


(a) *BRF*

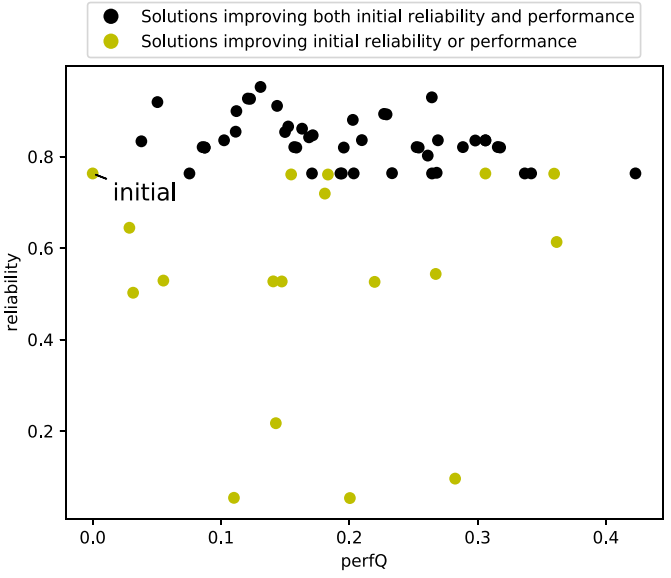


(b) *noBRF*

Fig. 11. The scatter plot of CoCoME Pareto frontiers while varying the fuzziness after 72 genetic evolutions with *BRF* and *noBRF* configurations.



(a) Train Ticket Booking Service



(b) CoCoME

Fig. 12. Solutions of the Pareto frontiers displayed according to their reliability and performance.

CoCoME. The case of CoCoME, in Fig. 12(b), strengthens the observations made for TTBS. In this case, the majority (*i.e.,* 74%) of the model alternatives improve both *perfQ* and *reliability* of the initial model. This is confirmed by the number of improving alternatives in the PF^{ref}: 38 out of 48. We got an improvement of the reliability up to 24%, which is smaller than TTBS but likely affected by the fact that, in this case, the starting model has higher initial reliability (*i.e.,* 0.75). Instead, the performance improvement is higher, *i.e.,* up to 42%.

Discussion. The set of model alternatives, which have been found while answering to RQ1, are characterized by a neat improvement of two quality attributes: *reliability* and *perfQ*. This result could be fundamental for designers, as they could do further analysis or use the model as a starting point in subsequent stages of the development process.

Our experimentation shows that, our approach can find design alternatives characterized by a significant improvement of both reliability and performance.

6.3. RQ3

RQ3: What type of refactoring actions are more likely to lead to better solutions?

With this research question, we investigate whether some refactoring actions are more likely to be selected than others in the Pareto optimal front during the optimization process. This could potentially lead to more general insights on the effectiveness of specific types of refactoring actions to improve the considered objectives.

Train Ticket Booking Service. Table 7 reports the share of refactoring types for Train Ticket Booking Service. Each row represents a configuration (*i.e.,* an experiment) with a different combination of *BRF*, *maxeval*,

Table 7
Share of refactoring types in Train Ticket.

brf	maxeval	probpas	Clon	MO2N	MO2C	ReDe
no	72	0.00	31.77	42.71	12.50	13.02
no	72	0.55	39.58	47.40	2.60	10.42
no	72	0.80	31.25	53.12	9.90	5.73
no	72	0.95	34.38	28.12	18.23	19.27
no	82	0.00	56.25	27.08	2.60	14.06
no	82	0.55	36.98	39.06	17.71	6.25
no	82	0.80	23.96	51.04	11.98	13.02
no	82	0.95	42.71	30.73	23.44	3.12
no	102	0.00	42.71	30.73	16.15	10.42
no	102	0.55	35.94	27.08	17.71	19.27
no	102	0.80	40.10	30.21	14.58	15.10
no	102	0.95	25.00	58.85	10.94	5.21
yes	72	0.00	40.10	30.21	16.15	13.54
yes	72	0.55	37.50	36.98	14.06	11.46
yes	72	0.80	42.71	19.27	16.15	21.88
yes	72	0.95	49.48	37.50	13.02	0.00
yes	82	0.00	19.79	57.81	10.42	11.98
yes	82	0.55	39.06	36.98	22.40	1.56
yes	82	0.80	27.60	40.62	13.54	18.23
yes	82	0.95	43.75	34.90	20.31	1.04
yes	102	0.00	43.75	35.94	16.67	3.65
yes	102	0.55	41.15	25.00	9.38	24.48
yes	102	0.80	35.42	40.10	10.42	14.06
yes	102	0.95	54.17	22.92	16.67	6.25
Total			38.13	36.85	14.06	10.96

Table 8
Share of refactoring types in CoCoME.

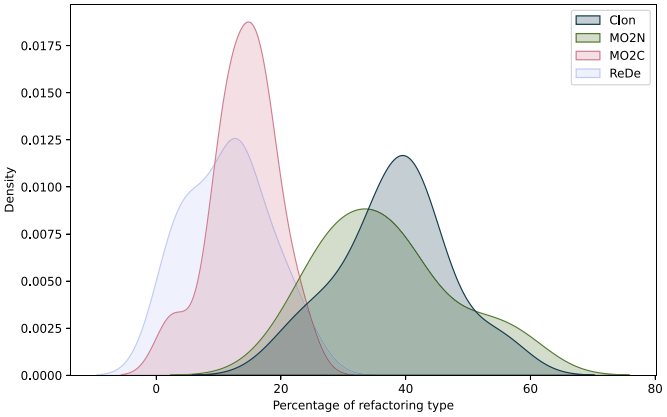
brf	maxeval	probpas	Clon	MO2N	MO2C	ReDe
no	72	0.00	30.21	37.50	19.79	12.50
no	72	0.55	54.69	24.48	12.50	8.33
no	72	0.80	42.19	32.81	18.75	6.25
no	72	0.95	45.83	37.50	9.90	6.77
no	82	0.00	43.23	25.52	17.19	14.06
no	82	0.55	48.96	27.60	12.50	10.94
no	82	0.80	37.50	41.67	10.42	10.42
no	82	0.95	53.12	28.12	5.73	13.02
no	102	0.00	20.83	36.46	17.71	25.00
no	102	0.55	44.27	28.12	20.31	7.29
no	102	0.80	55.73	27.08	1.04	16.15
no	102	0.95	56.25	28.65	13.54	1.56
yes	72	0.00	41.15	29.17	23.96	5.73
yes	72	0.55	38.02	32.81	20.83	8.33
yes	72	0.80	35.94	42.71	13.02	8.33
yes	72	0.95	61.98	23.44	10.94	3.65
yes	82	0.00	51.56	30.73	14.06	3.65
yes	82	0.55	41.67	33.85	18.75	5.73
yes	82	0.80	38.54	40.62	12.50	8.33
yes	82	0.95	44.27	26.56	16.67	12.50
yes	102	0.00	43.75	20.31	17.19	18.75
yes	102	0.55	66.67	6.25	20.31	6.77
yes	102	0.80	59.90	19.27	8.33	12.50
yes	102	0.95	61.46	16.67	8.33	13.54
Total			46.57	29.08	14.34	10.00

and *probpas*. The rightmost four columns represent the refactoring action types that we have considered in our approach. The last row shows the percentages computed over all the configurations.

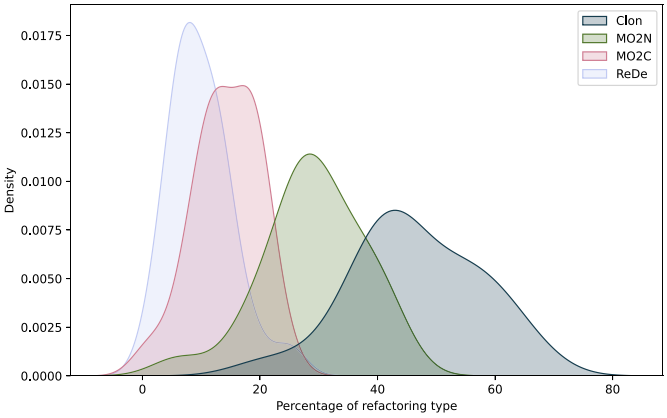
It is evident that the genetic algorithms prefer to select certain types of refactorings. *MO2C* and *Clon* are clearly more likely to be selected, with a slight preference for *Clon* in most configurations and, consequently, on average across all configurations. These refactorings are inherently very beneficial for the performance: cloning a component will frequently split the utilization in half, and moving an operation to a new component will not only reserve a node for a single operation, but will also relieve the original component of the load related to that operation. Also, they are unlikely to disrupt the reliability objective, since the new nodes will have the same probability of failure as the ones they are cloned from. Conversely, the *ReDe* refactoring may be advantageous for performance and reliability only when the component to be redeployed is sharing the current node with many other components, and this is not the case in the initial model. This is most probably the reason why the *ReDe* refactoring is considerably less likely to be selected, and there is even a configuration in which it was not selected in any Pareto solution (*BRF*: yes, *maxeval*: 72, *probpas*: 0.95).

CoCoME. Analogously, we report the share of refactoring actions for CoCoME in Table 8. The overall preferences in the selection of refactorings seem to be similar to the Train Ticket Booking Service case. However, we can notice an even stronger preference for the *Clon* refactoring. Since this refactoring largely decreases the utilization of nodes, it may be reasonable to conclude that, in the initial CoCoME model, some nodes with high utilization are preventing the performance to improve. While the *ReDe* refactoring is still the less selected one, there are no configurations in which at least one refactoring of this type does not contribute to Pareto solutions. However, in 13 configurations over a total of 24, the *ReDe* refactoring has a share below 10%.

Discussion. In both case studies, we can observe a common trend on preferring some refactoring types over other ones. In order to confirm that the trend is consistent, we show in Fig. 13 the density distributions of the shares of refactoring types across the different configurations. The order in which the distributions are shifted along the x-axis is the same in both cases, and their overlapping is somehow similar. This indicates that, on average, the refactoring types are selected with the same order of preference. We can also notice that, while in CoCoME the variability decreases together with the average percentage, in Train



(a) Train Ticket



(b) CoCoME

Fig. 13. Distributions of refactoring types among different configurations.

Ticket Booking Service the situation is less clear. A greater variability indicates that there are more chances that a change in the configuration will lead to a change in the selection preference of refactoring types, as it can be observed for *Clon* and *MO2N*. On the other hand, a narrow distribution means that configuration changes have little effect on the selection choice, as it happens for *MO2C* and *ReDe*. However, the refactorings that are more likely to be selected (*i.e., Clon* and *MO2N*) exhibit larger variability in both case studies, thus meaning that these refactorings are also the most variable ones from one configuration to another. This may indicate that, even if these two refactorings dominate, on average, the composition of solutions, the Pareto frontiers obtained by different configurations tend to be quite diverse.

Another aspect to consider is the influence of *BRF* on the choice of refactoring actions. While *BRF* clearly has a direct impact on the *#changes* objective, it looks like its presence is not enough to impose a different order of preference among the refactoring types. On the one hand, it could be expected that the *Clon* refactoring will be the most preferred because of its low *BRF* (1.23), but on the other hand the *MO2N* refactoring, that is consistently in the second place, has the highest value of *BRF*.

In an attempt to understand if there is a stronger relation between refactoring types and the objectives, we have also performed a multiple regression analysis. We tried to predict *perfQ*, *reliability*, and *#changes* using the refactoring types as predictors. The coefficients of determination (r^2) we obtained for each objective and for both case studies are very low. This means that the refactoring types are not suitable to explain most of the variability we observe in the objectives. Such a result might be the indication that, at least for the two case studies we considered, we are not able to derive general refactoring strategies to improve the objectives without going through the optimization process.

From our experimentation, we were able to establish an order of preference among refactoring types that is consistent in both case studies.

7. Threats to validity

The validity of our study can be affected by different threats described by the Wohlin et al. classification [46]. In the following, we detail each category by discussing the causes and motivations for each threat.

Construct validity. The way we have designed our problem and our experimentation might be affected by *Construct validity* threats. In particular, the role played by the architectural distance objective on the combination of refactoring actions might affect the selection of refactoring actions. However, we have studied the influence of our *BRF* in building PF^c in two different case studies, and it has coherently shown the ability to improve the overall quality of the non-dominated solutions in both cases. We will further investigate to what extent *BRF* could improve the overall quality with more accurate cost estimation, such as COCOMO [36], which might have as drawback the increase of the execution time for *BRF* estimation.

Another important aspect that might threaten our experimentation concerns the parameters of the initial UML model. For example, CoCoME showed higher initial reliability that might affect the search. However, in our experiments, it seems that TTBS and CoCoME initial configurations did not threaten the optimization process. We will further investigate how different initial UML model parameters could change the optimization results. We remark that changing a single model parameter means starting the optimization process on a different point of the solution space that might produce completely different results.

Internal validity. Our optimization approach might be affected by *internal validity* threats. There are high degrees of freedom on our settings. For example, the variations of genetic configurations, such as the $P_{crossover}$ probability, may produce PF^c with different quality solutions. Also, the problem configuration variations may also change our results. The degrees of freedom in our experimentation generate unfeasible brute force investigation of each suitable combination. For this reason, we limit the variability to subsets of problem configurations, as shown in Table 3. We also mitigate this threat by involving two different case studies derived from the literature, thus reducing biases in their construction.

A fruitful investigation will be on the length of the sequence of refactoring actions. At this stage, we fixed the length to four actions. It will be interesting to investigate how the length of the sequence affects results. At a glance, the longer the sequence, the farther the solutions can go from the initial one, and it means that having a long sequence of refactoring actions might be unfeasible because it generates different model alternatives.

External validity. Our results might be affected by *external validity* threats, as their generalization might be limited to some of the assumptions behind our approach.

In the first place, a threat might be represented by the use of a single modeling notation. We cannot generalize our results to other modeling notations, which could imply using a different portfolio of refactoring actions. The syntax and semantics of the modeling notation determine the amount and nature of refactoring actions that can be performed. However, we have adopted UML, which is the de facto standard in the software modeling domain. In general terms, this threat can be mitigated by porting the whole approach on a different modeling notation, but this is out of this paper scope.

Another threat might be found in the fact that we have validated our approach on two case studies. While the two case studies were selected from the available literature, they might not represent all the possible challenges that our approach could face in practice. Nonetheless, our results could presumably hold in all the cases in which the modeling assumptions described in Section 3.1 are met. Specifically, the performance antipattern detection and the refactoring actions are designed to rely on information coming from static, dynamic, and deployment views of the system. Without such information, even if in most cases the refactoring actions would still be applicable, they would not be as effective.

Finally, this study is limited to the use of a single algorithm. Therefore, our results are influenced by the ability of *NSGA-II* of exploring the solution space, given the objectives of our approach. While comparing the effectiveness of genetic algorithms in this context is out of the scope of this paper, we started investigating this issue [12,47], and we will continue in future work.

Conclusion validity. Our results might be affected by *Conclusion validity* threats, since our considerations might change with deeply-tuned parameters for the *NSGA-II*. Also, parameter configurations might threaten our conclusion. We did not perform an extensive tuning phase for the latter due to the long duration of each run, while we used common parameters for the *NSGA-II*, which should mitigate these threats. We can also soften this threat by employing other generic algorithms to generalize our results. Each algorithm will require its tuning phase, which is a clear drawback in execution time.

Another aspect that might affect our results is the estimation of the reference Pareto frontier (PF^{ref}). PF^{ref} is used for extracting the quality indicators as described in Section 6. We soften this threat by building the PF^{ref} overall our PF^c for each case study. Therefore, the reference Pareto should optimistically contain all non-dominated solutions across all configurations.

Takeaways. Model-based multi-objective refactoring optimization presents a variety of challenges that may jeopardize the validity of results.

Genetic algorithms contain a number of configuration options, to start. Every parameter assignment may have an effect on the outcomes quality. Indeed, there is opportunity for research direction here, since it would be impractical to evaluate every parameter combination. There have been studies on determining the (almost) ideal configuration of genetic algorithms in diverse contexts. We employed the standard genetic algorithm setup, such as the crossover probability [44]. However, it would be interesting to see which study applies to our situation as well. We plan to examine how different configurations affect the outcomes quality in future work.

The initial model setup is another factor taken into account. Studies that mix running data (such as traces) and model artifacts already exist to address this problem. There are plenty of shortcomings with these studies. We recently investigated the potential of model-based performance predictions when models are fed with running application data [48]. We discovered that if models take into account the confounding factors affecting application performance, such as network latency, they can anticipate the performance of the running application.

Moreover, the modeling notation affects how expressive the technique is. For instance, the use of a domain specific language to speed up design time could impair models expressiveness. Therefore, we chose to utilize UML, even though its broad general-purpose character is one of its disadvantages. With regard to the modeling and annotation practices in industry, the effort dedicated to these activities can largely vary depending on the field where industries work. As an example, automotive industries have adopted (since many decades) model-driven engineering approaches for designing their embedded software systems. For instance, Ameller et al. [49] provide an interesting study on the adoption in industrial contexts of modeling for sake of non-functional analysis.

Finally, regarding the applicability of the approach, it is difficult to establish a category of systems for which our approach would be better suited. Indeed, the only constraint that we require for its applicability is the usage of UML with the DAM [24] and MARTE [23] profiles. Obviously, such approach should be applied in systems where performance and reliability requirements have high priority. For example: distributed systems where reliable connections and timely response are main critical issues; embedded domains (e.g., automotive) where resources with limited hardware capability must guarantee high reliability. Centralized systems represent a further category of systems that may be subject to stringent performance requirement because, for example, a single host machine and its hardware resources must manage a complex software system.

8. Related work

In the last decade, software model multi-objective optimization studies have been introduced to optimize various quality attributes (e.g., reliability, and energy [11,50–52]) with different degrees of freedom in the model modification (*e.g.,* service selection [53,54]). A systematic literature review on model optimization can be found in [10]. We consider here, as related work, those approaches that directly involve multi-objective evolutionary algorithms, and the ones that exploit LQN as performance modeling notation [14,30,55,56].

We split this section in two subsections, namely *Software Architecture optimization* and *Layered Queueing Network approaches*. The partition is not strict, as it might happen that some studies fall in both conceptual areas. In order to prevent duplication, we chose to describe these studies in only one specific area.

8.1. Software architecture optimization

Menasce et al. have presented a framework for architectural design and quality optimization [57], where architectural patterns are used to support the search process (e.g., load balancing, fault tolerance). Two limitations affects the approach: the architecture has to be designed in a tool-related notation and not in a standard modeling language (as we do in this paper), and it uses equation-based analytical models for performance indices that could be too simple to capture architectural details and resource contention.

Aleti et al. [9] have presented an approach for modeling and analyzing AADL architectures [58]. They have also introduced a tool aimed at optimizing different quality attributes while varying the architecture deployment and the component redundancy. Our work relies on UML models and considers more complex refactoring actions, as well as different target attributes for the fitness function. Besides, we investigate the role of performance antipatterns in the context of many-objective software model refactoring optimization.

A recent work compares the ability of two different multi-objective optimization approaches to improve non-functional attributes [13], where randomized search rules have been applied to improve the software model. The study of Ni et al. is based on a specific modeling notation (*i.e.,* Palladio Component Model) and it has implicitly shown that the multi-objective optimization problem at model level is still an open challenge. They applied architectural tactics, which in general do not represent structured refactoring actions, to find optimal solutions. Conversely, we applied refactoring actions that change the structure of the initial model by preserving the original behavior. Another difference is the modeling notation, as we use UML with the goal of experimenting on a standard notation instead of a custom DSL.

Some authors of this paper have previously studied the sensitivity of multi-objective software model refactoring to configuration characteristics [12], where models are defined in Æmilia, which is a performance-oriented ADL. They compared two genetic algorithms in terms of Pareto frontiers quality. In this paper, we change the modeling notation from Æmilia to UML, and we add the reliability as a new objective. Both approaches provide a refactoring engine, however, in this paper, the refactoring engine offers more complex refactoring actions since UML is more expressive than Æmilia.

Etemaadi and Chaudron [59] presented an approach aimed at improving architecture quality attributes through genetic algorithms. The multi-objective optimization considers component-based architectures described through domain specific language (DSL), *i.e.,* AQOSA IR [50]. The architecture evaluations can be obtained by means of several notation, such as Queueing Network and Fault Tree. The genetic algorithm consider variation of designs (*e.g.,* number of hardware nodes) as objectives of the fitness function. The main difference between our approach and the one of Etemaadi and Chaudron is based on the types of the fitness function objectives. Yet, we used UML as the modeling notation instead of a DSL, and the LQN as the performance model.

8.2. Layered Queueing Network approaches

Koziolek et al. have presented PerOpteryx [14], *i.e.,* a performance-oriented multi-objective optimization problem. In PerOpteryx the optimization process is guided by tactics referring to component reallocation, faster hardware, and more hardware. The latter ones do not represent structured refactoring actions, as we intend in this paper. PerOpteryx supports architectures specified in Palladio Component Model [60] and produces, through model transformation, a LQN model for performance analysis.

Rago et al. have presented SQuAT [61], which is an extensible platform aimed at including flexibility in the definition of an architecture optimization problem. SQuAT supports models conforming to Palladio Component Model language, exploits LQN for performance evaluation, and PerOpteryx tactics for architectural changes. A main

difference of our approach with PerOpteryx and SQuAt is that we use the UML modeling notation. We moved a step ahead with respect PerOpteryx and SQuAT. Beyond the modeling notation, we introduced more complex refactoring actions, and we use different objectives, *e.g.,* performance antipatterns.

Model-to-model (M2M) transformations from UML to LQN notations have been presented in [30,55,56,62]. For example, Li et al. [30] presented a tool, namely Tulsa, aimed at enabling performance analysis of data intensive applications. Li et al. augmented UML models with the DICE profile, which allows expressing data intensive application domain specification. Also, they introduced a model-to-model transformation aimed at allowing a performance analysis through Layered Queueing Network. In contrast with these approaches, we present a novel M2M transformation mapping that employs UML Sequence Diagrams as the behavioral view of software architectures, instead of UML Activity Diagrams. UML Sequence Diagrams have two benefits: they are adopted more frequently than UML Activity Diagrams for software design [63], and they explicitly define method calls, while UML Activity Diagrams usually focus on workflows and processes. Therefore, our approach supports a more detailed behavioral representation in terms of time intervals between method calls.

9. Conclusions

In this work, we have used *NSGA-II* to optimize UML models with respect to performance and reliability properties, as well as the number of detected performance antipatterns and the architectural distance. We focused our study on the impact that performance antipatterns may have on the quality of optimal refactoring solutions. We studied the composition of refactoring actions, and how the architectural distance metric can help the approach to compute Pareto frontiers.

From our experimentation, we gathered interesting insights about the quality of the generated solutions and the role of performance antipatterns as an objective of the algorithm. In this regard, we showed that, by including the detection of performance antipatterns in the optimization process, we are able to obtain better solutions in terms of performance and reliability. Moreover, we also showed that, the more we increase the probability of detecting a performance antipattern using the fuzziness threshold, the better the quality of the refactoring solutions. In addition, we noticed that the *baseline refactoring factor* generally helps discovering better model alternatives. Another important aspect of our study was to ensure that our approach did not worsen the reliability of the initial model. In this respect, our experiments showed that we were in fact able to increase the reliability of model alternatives, with respect to the initial model, in the majority of cases.

As future work, we intend to tackle the threats to validity discussed before. In particular, we intend to investigate the influence of settings (*i.e.,* experiment and algorithm configurations) on the quality of Pareto frontiers. For example, we will investigate the impact of more dense populations in our analysis, in terms of computational time and quality of the computed Pareto frontiers (PF^c). Also, we are interested in the role played by *#changes*, and specifically in studying the effect of estimating the *baseline refactoring factor* through more complex cost model, such as COCOMO-II [36], on the combination of refactoring actions. A fruitful investigation will be on the length of the sequence of refactoring actions, which is currently fixed to four refactoring actions, and we intend to extend the refactoring actions portfolio, for example, by including fault tolerance refactoring actions [64]. We also intend to extend the reliability model to also take into account error propagation [65]. We will involve other genetic algorithms in our process to study the contribution of different optimization techniques within the software model refactoring.

We also planned to study how modeling outcomes could be verified and estimated on real-systems. As a first step to address this long-term study, we combined runtime traces (*i.e.,* traces from a running system) and modeling outcomes [48] and we found out that software models can help improve performance of software systems.

Another interesting aspect to investigate could be whether the refactoring actions proposed in the Pareto frontiers make sense form the point of view of the designer and within the established software development practices. Therefore, we plan on using visualization techniques to conduct a detailed analysis of the solutions resulting from the optimization process. Visualizing refactoring solutions also opens to a human-in-the-loop process, in which the designer could interactively drive the optimization towards acceptable solutions.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We publicly share the implementation of the approach https://github.com/SEALABQualityGroup/EASIER as well as the data gathered during the experimentation https://github.com/SEALABQualityGroup/2022-ist-replication-package.

Acknowledgments

Daniele Di Pompeo is supported by the Centre of EXcellence on Connected, Geo-Localized and Cybersecure Vehicle (EX-Emerge), funded by the Italian Government under CIPE resolution n. 70/2017 (Aug. 7, 2017). Michele Tucci is supported by the OP RDE project No. CZ.02.2.69/0.0/0.0/18_053/0016976 “International mobility of research, technical and administrative staff at the Charles University”.

References

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 2018.
[2] G. Bavota, M.D. Penta, R. Oliveto, Search based software maintenance: Methods and tools, in: T. Mens, A. Serebrenik, A. Cleve (Eds.), Evolving Software Systems, Springer, 2014, pp. 103–137, http://dx.doi.org/10.1007/978-3-642-45398-4_4.
[3] M. Kessentini, H.A. Sahraoui, M. Boukadoum, O. Benomar, Search-based model transformation by example, Softw. Syst. Model. 11 (2) (2012) 209–226, http://dx.doi.org/10.1007/s10270-010-0175-7.
[4] T. Mariani, S.R. Vergilio, A systematic review on search-based refactoring, Inf. Softw. Technol. 83 (2017) 14–34, http://dx.doi.org/10.1016/j.infsof.2016.11.009.
[5] A. Ouni, R.G. Kula, M. Kessentini, K. Inoue, Web service antipatterns detection using genetic programming, in: S. Silva, A.I. Esparcia-Alcázar (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, ACM, 2015, pp. 1351–1358, http://dx.doi.org/10.1145/2739480.2754724.
[6] A. Ouni, M. Kessentini, K. Inoue, M.Ó. Cinnéide, Search-based web service antipatterns detection, IEEE Trans. Serv. Comput. 10 (4) (2017) 603–617, http://dx.doi.org/10.1109/TSC.2015.2502595.
[7] A. Ramírez, J.R. Romero, S. Ventura, A survey of many-objective optimisation in search-based software engineering, J. Syst. Softw. 149 (2019) 382–395, http://dx.doi.org/10.1016/j.jss.2018.12.015.
[8] M. Ray, D.P. Mohapatra, Multi-objective test prioritization via a genetic algorithm, Innov. Syst. Softw. Eng. 10 (4) (2014) 261–270, http://dx.doi.org/10.1007/s11334-014-0234-2.
[9] A. Aleti, S. Björnander, L. Grunske, I. Meedeniya, ArcheOpterix: An extendable tool for architecture optimization of AADL models, in: ICSE 2009 Workshop on Model-Based Methodologies for Pervasive and Embedded Software, MOMPES 2009, May 16, 2009, Vancouver, Canada, IEEE Computer Society, 2009, pp. 61–71, http://dx.doi.org/10.1109/MOMPES.2009.5069138.
[10] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, I. Meedeniya, Software architecture optimization methods: A systematic literature review, IEEE Trans. Softw. Eng. 39 (5) (2013) 658–683, http://dx.doi.org/10.1109/TSE.2012.64.
[11] A. Martens, H. Koziolek, S. Becker, R.H. Reussner, Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms, in: A. Adamson, A.B. Bondi, C. Juiz, M.S. Squillante (Eds.), Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering, San Jose, California, USA, January 28-30, 2010, ACM, 2010, pp. 105–116, http://dx.doi.org/10.1145/1712605.1712624.

[12] V. Cortellessa, D. Di Pompeo, Analyzing the sensitivity of multi-objective software architecture refactoring to configuration characteristics, Inf. Softw. Technol. 135 (2021) 106568, http://dx.doi.org/10.1016/j.infsof.2021.106568.

[13] Y. Ni, X. Du, P. Ye, L.L. Minku, X. Yao, M. Harman, R. Xiao, Multi-objective software performance optimisation at the architecture level using randomised search rules, Inf. Softw. Technol. 135 (2021) 106565, http://dx.doi.org/10.1016/j.infsof.2021.106565.

[14] A. Koziolek, H. Koziolek, R.H. Reussner, PerOpteryx: automated application of tactics in multi-objective software architecture optimization, in: I. Crnkovic, J.A. Stafford, D.C. Petriu, J. Happe, P. Inverardi (Eds.), 7th International Conference on the Quality of Software Architectures, QoSA 2011 and 2nd International Symposium on Architecting Critical Systems, ISARCS 2011. Boulder, CO, USA, June 20-24, 2011, Proceedings, ACM, 2011, pp. 33–42, http://dx.doi.org/10.1145/2000259.2000267.

[15] K. Deb, S. Agrawal, A. Pratap, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197, http://dx.doi.org/10.1109/4235.996017.

[16] J.E. Neilson, C.M. Woodside, D.C. Petriu, S. Majumdar, Software bootlenecking in client-server systems and rendezvous networks, IEEE Trans. Softw. Eng. 21 (9) (1995) 776–782, http://dx.doi.org/10.1109/32.464543.

[17] V. Cortellessa, H. Singh, B. Cukic, Early reliability assessment of UML based software models, in: Third International Workshop on Software and Performance, WOSP@ISSTA 2002, July 24-26, 2002, Rome, Italy, ACM, 2002, pp. 302–309, http://dx.doi.org/10.1145/584369.584415.

[18] D. Arcelli, V. Cortellessa, D. Di Pompeo, Performance-driven software model refactoring, Inf. Softw. Technol. 95 (2018) 366–397, http://dx.doi.org/10.1016/j.infsof.2017.09.006.

[19] D. Arcelli, V. Cortellessa, M. D'Emidio, D. Di Pompeo, EASIER: an evolutionary approach for multi-objective software ArchItecturE refactoring, in: IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018, IEEE Computer Society, 2018, pp. 105–114, http://dx.doi.org/10.1109/ICSA.2018.00020.

[20] C.U. Smith, L.G. Williams, Software performance antipatterns, in: Second International Workshop on Software and Performance, WOSP 2000, Ottawa, Canada, September 17-20, 2000, ACM, 2000, pp. 127–136, http://dx.doi.org/10.1145/350391.350420.

[21] C.U. Smith, L.G. Williams, Software performance AntiPatterns; common performance problems and their solutions, in: 27th International Computer Measurement Group Conference, Anaheim, CA, USA, December 2-7, 2001, Computer Measurement Group, 2001, pp. 797–806.

[22] C.U. Smith, L.G. Williams, More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot, in: 29th International Computer Measurement Group Conference, 2003, pp. 717–725.

[23] O.M. Group, A UML Profile For MARTE: Modeling and Analysis of Real-Time Embedded Systems, Object Management Group, 2008, URL: http://www.omg.org/omgmarte/.

[24] S. Bernardi, J. Merseguer, D.C. Petriu, A dependability profile within MARTE, Softw. Syst. Model. 10 (3) (2011) 313–336, http://dx.doi.org/10.1007/s10270-009-0128-1.

[25] V. Cortellessa, D. Di Pompeo, V. Stoico, M. Tucci, On the impact of performance antipatterns in multi-objective software model refactoring optimization, in: M.T. Baldassarre, G. Scanniello, A. Skavhaug (Eds.), 47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2021, Palermo, Italy, September 1-3, 2021, IEEE, 2021, pp. 224–233, http://dx.doi.org/10.1109/SEAA53835.2021.00036.

[26] D. Di Pompeo, M. Tucci, A. Celi, R. Eramo, A microservice reference case study for design-runtime interaction in MDE, in: A. Bagnato, H. Brunelière, L.B. no, R. Eramo, A. Gómez (Eds.), STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop Co-Located with Software Technologies: Applications and Foundations (STAF 2019), Eindhoven, the Netherlands, July 15 - 19, 2019, CEUR-WS.org, 2019, pp. 23–32, URL: http://ceur-ws.org/Vol-2405/06_paper.pdf.

[27] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, D. Ding, Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study, IEEE Trans. Softw. Eng. 47 (2) (2021) 243–260, http://dx.doi.org/10.1109/TSE.2018.2887384.

[28] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, C. Pfaller, Cocome - the common component modeling example, in: The Common Component Modeling Example: Comparing Software Component Models, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 16–53, http://dx.doi.org/10.1007/978-3-540-85289-6_3.

[29] U. Mansoor, M. Kessentini, M. Wimmer, K. Deb, Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm, Softw. Qual. J. 25 (2) (2017) 473–501, http://dx.doi.org/10.1007/s11219-015-9284-4.

[30] C. Li, T. Altamimi, M.H. Zargari, G. Casale, D.C. Petriu, Tulsa: A tool for transforming UML to layered queueing networks for performance analysis of data intensive applications, in: N. Bertrand, L. Bortolussi (Eds.), Quantitative Evaluation of Systems - 14th International Conference, QEST 2017, Berlin, Germany, September 5-7, 2017, Proceedings, Springer, 2017, pp. 295–299, http://dx.doi.org/10.1007/978-3-319-66335-7_18.

[31] C.U. Smith, L.G. Williams, Software performance engineering, in: L. Lavagno, G. Martin, B. Selic (Eds.), UML for Real - Design of Embedded Real-Time Systems, Kluwer, 2003, pp. 343–365, http://dx.doi.org/10.1007/0-306-48738-1_16.

[32] V. Cortellessa, A. Di Marco, C. Trubiani, An approach for modeling and detecting software performance antipatterns based on first-order logics, Softw. Syst. Model. 13 (1) (2014) 391–432, http://dx.doi.org/10.1007/s10270-012-0246-z.

[33] D. Arcelli, V. Cortellessa, C. Trubiani, Performance-based software model refactoring in fuzzy contexts, in: A. Egyed, I. Schaefer (Eds.), Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, Springer, 2015, pp. 149–164, http://dx.doi.org/10.1007/978-3-662-46675-9_10.

[34] D. Arcelli, V. Cortellessa, D. Di Pompeo, Automating performance antipattern detection and software refactoring in UML models, in: X. Wang, D. Lo, E. Shihab (Eds.), 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019, IEEE, 2019, pp. 639–643, http://dx.doi.org/10.1109/SANER.2019.8667967.

[35] D. Arcelli, V. Cortellessa, D. Di Pompeo, A metamodel for the specification and verification of model refactoring actions, in: A. Ouni, M. Kessentini, M.O. Cinnéide (Eds.), Proceedings of the 2nd International Workshop on Refactoring, IWoR@ASE 2018, Montpellier, France, September 4, 2018, IWoR@ACM, 2018, pp. 14–21, http://dx.doi.org/10.1145/3242163.3242167.

[36] B.W. Boehm, C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D.J. Reifer, B. Steece, SoftWare Cost Estimation With COCOMO II, Prentice Hall Press, 2009.

[37] A. Trendowicz, Software Cost Estimation, Benchmarking, and Risk Assessment: The Software Decision-Makers' Guide to Predictable Software Development, Springer Science & Business Media, 2013.

[38] A.J. Nebro, J.J. Durillo, M. Vergne, Redesigning the jmetal multi-objective optimization framework, in: S. Silva, A.I. Esparcia-Alcázar (Eds.), Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings, ACM, 2015, pp. 1093–1100, http://dx.doi.org/10.1145/2739482.2768462.

[39] S. Ali, P. Arcaini, D. Pradhan, S.A. Safdar, T. Yue, Quality indicators in search-based software engineering: An empirical evaluation, ACM Trans. Softw. Eng. Methodol. 29 (2) (2020) 10:1–10:29, http://dx.doi.org/10.1145/3375636.

[40] A. Zhou, Y. Jin, Q. Zhang, B. Sendhoff, E.P.K. Tsang, Combining model-based and genetics-based offspring generation for multi-objective optimization using a convergence criterion, in: IEEE International Conference on Evolutionary Computation, CEC 2006, Part of WCCI 2006, Vancouver, BC, Canada, 16-21 July 2006, IEEE, 2006, pp. 892–899, http://dx.doi.org/10.1109/CEC.2006.1688406.

[41] H. Ishibuchi, H. Masuda, Y. Nojima, Sensitivity of performance evaluation results by inverted generational distance to reference points, in: IEEE Congress on Evolutionary Computation, CEC 2016, Vancouver, BC, Canada, July 24-29, 2016, IEEE, 2016, pp. 1107–1114, http://dx.doi.org/10.1109/CEC.2016.7743912.

[42] E. Zitzler, L. Thiele, Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach, IEEE Trans. Evol. Comput. 3 (4) (1999) 257–271, http://dx.doi.org/10.1109/4235.797969.

[43] E. Zitzler, L. Thiele, M. Laumanns, C.M. Fonseca, V.G. da Fonseca, Performance assessment of multiobjective optimizers: an analysis and review, IEEE Trans. Evol. Comput. 7 (2) (2003) 117–132, http://dx.doi.org/10.1109/TEVC.2003.810758.

[44] A. Arcuri, G. Fraser, Parameter tuning or default values? An empirical investigation in search-based software engineering, Empir. Softw. Eng. 18 (3) (2013) 594–623, http://dx.doi.org/10.1007/s10664-013-9249-9.

[45] A. Arcuri, G. Fraser, On parameter tuning in search based software engineering, in: M.B. Cohen, M.O. Cinnéide (Eds.), Search Based Software Engineering - Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings, Springer, 2011, pp. 33–47, http://dx.doi.org/10.1007/978-3-642-23716-4_6.

[46] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, Experimentation in software engineering, Springer, 2012, http://dx.doi.org/10.1007/978-3-642-29044-2.

[47] D. Di Pompeo, M. Tucci, Search budget in multi-objective refactoring optimization: a model-based empirical study, in: 48th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2022, IEEE, 2022, pp. 406–413, http://dx.doi.org/10.1109/SEAA56994.2022.00070.

[48] V. Cortellessa, D. Di Pompeo, R. Eramo, M. Tucci, A model-driven approach for continuous performance engineering in microservice-based systems, J. Syst. Softw. 183 (2022) 111084, http://dx.doi.org/10.1016/j.jss.2021.111084.

[49] D. Ameller, X. Franch, C. Gómez, S. Martínez-Fernández, J. Araújo, S. Biffl, J. Cabot, V. Cortellessa, D.M. Fernández, A. Moreira, H. Muccini, A. Vallecillo, M. Wimmer, V. Amaral, W. Böhm, H. Brunelière, L. Burgueño, M. Goulão, S. Teufl, L. Berardinelli, Dealing with non-functional requirements in model-driven development: A survey, IEEE Trans. Softw. Eng. 47 (4) (2021) 818–835, http://dx.doi.org/10.1109/TSE.2019.2904476.

[50] R. Li, R. Etemaadi, M.T.M. Emmerich, M.R.V. Chaudron, An evolutionary multiobjective optimization approach to component-based software architecture design, in: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2011, New Orleans, la, USA, 5-8 June, 2011, IEEE, 2011, pp. 432–439, http://dx.doi.org/10.1109/CEC.2011.5949650.

[51] I. Meedeniya, B. Buhnova, A. Aleti, L. Grunske, Architecture-driven reliability and energy optimization for complex embedded systems, in: G.T. Heineman, J. Kofron, F. Plasil (Eds.), Research Into Practice - Reality and Gaps, 6th International Conference on the Quality of Software Architectures, QoSA 2010, Prague, Czech Republic, June 23 - 25, 2010. Proceedings, Springer, 2010, pp. 52–67, http://dx.doi.org/10.1007/978-3-642-13821-8_6.

[52] A. Martens, D. Ardagna, H. Koziolek, R. Mirandola, R.H. Reussner, A hybrid approach for multi-attribute QoS optimisation in component based software systems, in: G.T. Heineman, J. Kofron, F. Plasil (Eds.), Research Into Practice - Reality and Gaps, 6th International Conference on the Quality of Software Architectures, QoSA 2010, Prague, Czech Republic, June 23 - 25, 2010. Proceedings, Springer, 2010, pp. 84–101, http://dx.doi.org/10.1007/978-3-642-13821-8_8.

[53] F. Rosenberg, M.B. Müller, P. Leitner, A. Michlmayr, A. Bouguettaya, S. Dustdar, Metaheuristic optimization of large-scale QoS-aware service compositions, in: 2010 IEEE International Conference on Services Computing, SCC 2010, Miami, Florida, USA, July 5-10, 2010, IEEE Computer Society, 2010, pp. 97–104, http://dx.doi.org/10.1109/SCC.2010.58.

[54] V. Cardellini, E. Casalicchio, V. Grassi, F.L. Presti, R. Mirandola, Qos-driven runtime adaptation of service oriented architectures, in: H. van Vliet, V. Issarny (Eds.), Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, the Netherlands, August 24-28, 2009, ACM, 2009, pp. 131–140, http://dx.doi.org/10.1145/1595696.1595718.

[55] C.M. Woodside, D.C. Petriu, D.B. Petriu, H. Shen, T. Israr, J. Merseguer, Performance by unified model analysis (PUMA), in: Proceedings of the Fifth International Workshop on Software and Performance, WOSP 2005, Palma, Illes Balears, Spain, July 12-14, 2005, ACM, 2005, pp. 1–12, http://dx.doi.org/10.1145/1071021.1071022.

[56] T. Altamimi, D.C. Petriu, Incremental change propagation from UML software models to LQN performance models, in: M. Mindel, K.A. Lyons, J. Wigglesworth (Eds.), Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering, CASCON 2017, Markham, Ontario, Canada, November 6-8, 2017, IBM / ACM, 2017, pp. 120–131, URL: http://dl.acm.org/citation.cfm?id=3172810.

[57] D.A. Menascé, J.M. Ewing, H. Gomaa, S. Malek, J.P. Sousa, A framework for utility-based service oriented design in SASSY, in: A. Adamson, A.B. Bondi, C. Juiz, M.S. Squillante (Eds.), Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering, San Jose, California, USA, January 28-30, 2010, ACM, 2010, pp. 27–36, http://dx.doi.org/10.1145/1712605.1712612.

[58] P.H. Feiler, D.P. Gluch, Model-Based Engineering with AADL - an Introduction to the SAE Architecture Analysis and Design Language, SEI series in software engineering, Addison-Wesley, 2012, URL: http://www.pearsoned.co.uk/bookshop/detail.asp?item=100000000518651.

[59] R. Etemaadi, M.R.V. Chaudron, New degrees of freedom in metaheuristic optimization of component-based systems architecture: Architecture topology and load balancing, Sci. Comput. Program. 97 (2015) 366–380, http://dx.doi.org/10.1016/j.scico.2014.06.012.

[60] S. Becker, H. Koziolek, R.H. Reussner, The palladio component model for model-driven performance prediction, J. Syst. Softw. 82 (1) (2009) 3–22, http://dx.doi.org/10.1016/j.jss.2008.03.066.

[61] A. Rago, S.A. Vidal, J.A. Diaz-Pace, S. Frank, A. van Hoorn, Distributed quality-attribute optimization of software architectures, in: Proceedings of the 11th Brazilian Symposium on Software Components, Architectures and Reuse, SBCARS 2017, Fortaleza, CE, Brazil, September 18 - 19, 2017, ACM, 2017, pp. 7:1–7:10, http://dx.doi.org/10.1145/3132498.3132509.

[62] T. Altamimi, M.H. Zargari, D.C. Petriu, Performance analysis roundtrip: automatic generation of performance models and results feedback using cross-model trace links, in: M. Mindel, B. Jones, H.A. Müller, V. Onut (Eds.), Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, CASCON 2016, Toronto, Ontario, Canada, October 31 - November 2, 2016, IBM / ACM, 2016, pp. 208–217, URL: http://dl.acm.org/citation.cfm?id=3049899.

[63] J. Erickson, K. Siau, Can UML be simplified? Practitioner use of UML in separate domains, in: E. Proper, T.A. Halpin, J. Krogstie (Eds.), Proceedings of the 12th International Workshop on Exploring Modeling Methods for Systems Analysis and Design, EMMSAD 2008, Held in Conjunction with the 19th Conference on Advanced Information Systems (CAiSE 2007), Trondheim, Norway, 11-15 June, 2007, CEUR-WS.org, 2007, pp. 81–90, URL: http://ceur-ws.org/Vol-365/paper9.pdf.

[64] V. Cortellessa, R. Eramo, M. Tucci, From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement, Inf. Softw. Technol. 127 (2020) 106362, http://dx.doi.org/10.1016/j.infsof.2020.106362.

[65] V. Cortellessa, V. Grassi, A modeling approach to analyze the impact of error propagation on reliability of component-based systems, in: H.W. Schmidt, I. Crnkovic, G.T. Heineman, J.A. Stafford (Eds.), Component-Based Software Engineering, 10th International Symposium, CBSE 2007, Medford, MA, USA, July 9-11, 2007, Proceedings, Springer, 2007, pp. 140–156, http://dx.doi.org/10.1007/978-3-540-73551-9_10.