Analyzing the sensitivity of multi-objective software architecture refactoring to configuration characteristics

Vittorio Cortellessa, Daniele Di Pompeo *

Department of Information Engineering, Computer Science and Mathematics, University of L’Aquila, Italy

A R T I C L E   I N F O

Keywords:
- Search-based software engineering
- Automated refactoring
- Software quality
- Multi-objective optimization
- Genetic algorithms
- Software performance engineering
- Software performance antipatterns

A B S T R A C T

Context: Software architecture refactoring can be induced by multiple reasons, such as satisfying new functional requirements or improving non-functional properties. Multi-objective optimization approaches have been widely used in the last few years to introduce automation in the refactoring process, and they have revealed their potential especially when quantifiable attributes are targeted. However, the effectiveness of such approaches can be heavily affected by configuration characteristics of the optimization algorithm, such as the composition of solutions.

Objective: In this paper, we analyze the behavior of EASIER, which is an Evolutionary Approach for Software architecture Refactoring, while varying its configuration characteristics, with the objective of studying its potential to find near-optimal solutions under different configurations.

Method: In particular, we use two different solution space inspection algorithms (i.e., NSGA – II and SPEA2) while varying the genome length and the solution composition.

Results: We have conducted our experiments on a specific case study modeled in Æmilia ADL, on which we have shown the ability of EASIER to identify performance-critical elements in the software architecture where refactoring is worth to be applied. Beside this, from the comparison of multi-objective algorithms, NSGA – II has revealed to outperform SPEA2 in most of cases, although the latter one is able to induce more diversity in the proposed solutions.

Conclusion: Our results show that the EASIER thoroughly automated process for software architecture refactoring allows to identify configuration contexts of the evolutionary algorithm in which multi-objective optimization more effectively finds near-optimal Pareto solutions.

1. Introduction

Multi-objective optimization techniques have been successfully applied to many software architecture problems in the last decades [1–7]. These techniques are evidently more effective on problems whose objectives can be expressed through quantifiable metrics. Problems related to non-functional aspects of software architecture evidently fit into this category, as witnessed by the vast literature in this domain [8–10]. Many of such approaches are based on evolutionary algorithms [11] that allow to search the solution space by (re-)combining solutions. Architecture refactoring is a task that may be induced in different scenarios, such as the introduction of additional requirements, the adaptation to new execution contexts, the degradation of non-functional properties. Although the identification of optimal architectural refactoring is a non trivial task, mostly due to the large space of solutions, there is still lack of automated support to software architects in this context.

We have recently introduced EASIER, which is an Evolutionary Approach for Software architecture Refactoring, aimed at optimizing refactoring actions on the basis of metrics related to the performance and to the distance from the initial architecture [12]. EASIER deals with genomes that represent sequences of refactoring actions aimed at leading to optimal architectural alternatives starting from an initial one. A common aspect of similar approaches introduced before EASIER is that they search among architectural alternatives, without considering the operational aspects induced by architectural refactoring. Instead, with software architecture gaining relevance across the whole lifecycle (even after software release), the path of architectural refactoring assumes a high relevance that we have considered as a main aspect of EASIER. We have also introduced in this context the knowledge dwelling in performance antipatterns [13] as a support to the search process. Performance antipatterns are, in fact, well-known bad practices that induce performance degradation.

* Corresponding author.

E-mail addresses: vittorio.cortellessa@univaq.it (V. Cortellessa), daniele.dipompeo@univaq.it (D. Di Pompeo).

https://doi.org/10.1016/j.infsof.2021.106568

Received 21 July 2020; Received in revised form 5 February 2021; Accepted 3 March 2021

Available online 13 March 2021

0950-5849/© 2021 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).
However, the effectiveness of evolutionary approaches to find satisfactory solutions depends on their configuration that is defined before the searching process starts, which is intended as the set of values assigned to the configuration characteristics, such as the size of initial population and the genome length. In this paper, we push EASIER a step ahead in the direction of analyzing the sensitivity of its effectiveness to variations of the following categories of configuration characteristics:

- Search algorithm — Beside the NSGA-II algorithm adopted in [12], we introduce here SPEA2 (Strength Pareto Evolutionary Algorithm) [14].
- General parameters — We vary the genome length and composition, as we study the effectiveness of our approach while varying the number and weights of refactoring actions.

We focus our experimentation on answering to the following research questions:

- \( RQ_1 \): Which evolutionary algorithm proposes more interesting near-Pareto solutions on our multi-objective optimization problem between NSGA – II and SPEA2, while varying their configuration characteristics? In particular:
  - \( RQ_{11} \): What is the role of the cloning action within obtained solutions?
  - \( RQ_{12} \): What is the role of change rate action within obtained solutions?
- \( RQ_2 \): Which evolutionary algorithm better inspects the solution space on our multi-objective optimization problem between NSGA – II and SPEA2, while varying the configuration characteristics?
- \( RQ_3 \): Is EASIER able to reach a desired solution independently of the search starting point within the solution space?

Based on our experimentation, we can state that EASIER is able to identify performance-critical (sequences of) actions within Æmilia architectures. Moreover, EASIER can choose an optimal configuration, by exploiting well-known quality indicators, among all possible ones. Such configuration enables to find near-optimal Pareto frontiers with respect to performance indices more effectively. Finally, EASIER also allows analyzing the role of refactoring actions and their correlation with the goodness of near-optimal Pareto frontiers.

The paper is structured as follows: in Section 2 we report the related work; Section 3 introduces background notations; Section 4 describes the EASIER functionalities; Section 5 presents the case study and the EASIER tool, while Section 6 reports a detailed discussion on results; Section 7 describes the approach’s threats to validity, and finally Section 8 concludes the paper.

2. Related work

With the continuous evolution of software systems even after release, automation in refactoring has become a critical need along the whole development process [15]. In fact, many studies have been conducted in the context of model-based software refactoring (see, e.g., [16–18]).

However, finding the best sequence of refactoring actions to be applied to a software artifact in order to optimize its quality with respect to a set of objectives, is a problem known to be computationally hard, due to the typically huge space of feasible solutions [2]. Hence, its exhaustive solution can require large computational time even for small-sized software artifacts. One way of addressing this issue consists in formulating the problem as a search-based problem and tackling it via meta-heuristics (e.g., evolutionary algorithms) that are able to compute sets of refactoring actions that are optimal in a Pareto sense. To this regard, a number of studies have demonstrated the effectiveness of this strategy [19–22].

Several evolutionary algorithms have been introduced, in the last decade, for software architecture multi-objective optimization with respect to various quality attributes (e.g., reliability, performance or energy [9,23–25]) and with different degrees of freedom to modify the architecture (e.g., service selection, composition or deployment [26, 27]). A systematic literature review on architecture optimization can be found in [8].

An interesting contribution in this direction was given in [9,28], where an evolutionary algorithm for architecture optimization is guided by tactics, which are common practices applied by experienced software engineers when designing an architecture (e.g., fast pathing, caching). Out of a dozen of defined tactics, the authors have implemented three of them to observe their impact on the search algorithm. However, they refer to component reallocation, faster hardware and more hardware, so they do not represent structured refactoring actions, as we intend to do in this paper. Moreover, their approach starts from an architecture specified in Palladio Component Model [29] and produces, through model transformation, a Layered Queueing Network for sake of performance analysis.

On the basis of PerOpteryx tactics introduce in [28], Rango et al. have presented an extensible platform aimed at introducing flexibility in the definition of an architecture optimization problem [30]. They have implemented SQuAT, which allows the architect to define her specific roles (i.e., tactics) to improve the flexibility within the architecture optimization driven by non-functional properties. Furthermore, SQuAT supports models conforming to Palladio Component Model language, and exploits Layered Queueing Networks for performance evaluation. Instead, our approach works entirely within the Æmilia ADL environment, hence it is not subject to changes of notation that may induce inaccuracies in the performance model.

Another relevant approach has been introduced in [31], where architectural patterns are used to support the searching process (e.g., load balancing, fault tolerance). The authors introduce a whole framework for architectural design and quality optimization. This approach has two limitations, that are: the architecture has to be designed in a tool-related notation and not in a common ADL (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.

Another architecture optimization platform, namely AQOSA, was presented by Etemaadi et al. [23]. In particular, AQOSA is aimed at finding near-Pareto frontiers by exploiting ROBOCOP modeling language [32]. Similarly to our approach, AQOSA compares three evolutionary algorithms (i.e., NSGA – II, SPEA2, SMS-EMOA [33]) to determine which one produces better frontiers in terms of conflicting objectives. AQOSA supports three objectives, as EASIER does, but they are specific of hardware configuration. Conversely, we have implemented refactoring actions aimed at changing either the hardware configuration (i.e., rate changing) or the structure of the architecture by preserving the original behavior (i.e., component cloning).

An approach taking place in a unique environment for modeling and analysis has appeared in [10]. A tool is introduced, based on AADL [34], aimed at optimizing different quality attributes while varying the architecture deployment and the component redundancy. Our paper works on a different ADL, that is Æmilia, and it introduces more complex refactoring actions, as well as different target attributes for the fitness function. In addition, we investigate the role of performance antipatterns in this context.

Hence, the major novelties of EASIER, with respect to the existing literature, are that: (i) it works within a single environment for architectural modeling and analysis (i.e., Æmilia), (ii) it defines novel degrees of freedom aimed at representing the operational aspects of architectural refactoring, (iii) it introduces new attributes for the fitness function, that are a performance quality indicator and an architectural distance metrics, (iv) it starts to investigate the role of performance
antipatterns in this context, and (v) it has been conceived to host different ADLs.

The Æmilia syntax is based on formal constructs that somehow limit the size and complexity of architecture details that can be realistically represented within an Æmilia specification. However, this aspect is common to all ADLs that aim to be not only conceptually descriptive, but also parsable for sake of correctness or, like in the Æmilia case, for sake of performance analysis. Hence, formalism is a compulsory cost that architects have to pay if they require the architecture specification to be straightforwardly analyzed. In some cases, like the one illustrated in Section 6.1.1, the language syntax can also constrain the type of refactoring actions that can be applied for improving the architecture.

The choice of such an ADL within the EASIER context has been mostly induced by point (i) named above, i.e., an unique modeling and analysis environment. In fact, the iterative EASIER process requires automation in the analysis of different (possibly refactored) architecture specifications, and the Æmilia runtime environment enables this automation without needing to move towards a different modeling notation.

Alternative modeling languages that could empower more freedom to architectural designers, like UML, need a major effort to be adopted within an iterative automated process like the EASIER one. Indeed, for example, performance cannot be directly analyzed on an UML model, because even though performance parameters and index can be annotated (e.g., through the UML MARTE profile¹), the model needs to be transformed in a performance-specific notation (like Queueing Networks) to be analyzed. In addition, on the way back, analysis results have to be fed back onto the UML model for sake of searching optimal refactoring actions. These additional steps should be executed at each EASIER iteration, thus jeopardizing the soundness of the whole process, due to potential faults in the translation and in the resulting report, as well as the process execution time.

3. Background

This section introduces background concepts related to Æmilia, which is the Architectural Description Language on which EASIER is based, to NSGA-II and SPEA2, that are the evolutionary algorithms adopted in EASIER, and to the refactoring feasibility definition that we adopt in EASIER.

3.1. Æmilia

The performance analysis of a software architecture is a particularly complex process because performance metrics emerge from the combination of several characteristics, i.e., static, dynamic, deployment and – in some cases – environmental ones. Hence, very few ADLs embed constructs to specify performance parameters, and even fewer ones provide tools to analyze performance within the same ADL environment natively. In most cases, instead, performance models are expressed in different stochastic notations (like Queueing Networks or Petri Nets), thus they have to be generated from architectural specifications through model transformations [35].

These transformations introduce an additional step that could affect the process efficiency and correctness. We opted for avoiding these potential problems in this paper, because we intended to focus on the sensitivity analysis of EASIER to the configuration characteristics. Hence, as in [12], we kept EASIER based on Æmilia, that is an Architectural Description Language based on a Process Algebra that allows to in-place evaluate non-functional properties, such as performance, without needing to transform the architectural description in a different notation [36].

Basically, an Æmilia specification is textually described as shown in the example Listing 1, where:

```
Listing 1: An excerpt of an Æmilia architecture

1 ARCHI_TYPE fta(
2   const rate workload := 190.815,
3   const rate wsn_rate := 1.457,
4   const rate channel_rate := 2.562,
5   const rate fta_rate := 6.634,
6   [...]
7   const rate db_rate := 0.538,
8   const rate dec_rate := 0.139)
9 ARCHI_ELEM_TYPES
10 ELEM_TYPE
11 Wsn_Type(workload, wsn_rate)
12 BEHAVIOR
13 SendData(void;void) = <generate_data, exp(wsn_rate)>
14 <send_data, exp(wsn_rate)>. SendData() 
15 INPUT_INTERACTIONS
16 UNI generate_data
17 OUTPUT_INTERACTIONS
18 UNI send_data
19 [...]
20 ARCHI_ELEM_INSTANCES
21 WSN : Wsn_Type(workload, wsn_rate);
22 [...]
23 ARCHI_INTERACTIONS
24 WSN.generate_data;
25 [...]
26 ARCHI_ATTACHMENTS
27 FROM WSN.send_data TO CHN.receive_data;
28 [...]
29 END
```

• Constant declaration area (lines 2–8) contains the “global” variables;
• Architectural types (lines 9–19) in which components and their internal behavior are designed;
• Architectural topology (lines 20–29) in which connection among components are defined.

In particular, a component is described through the ELEM_TYPE tag that allows designing its structure and behavior. The structure is defined through INPUT_INTERACTION, and OUTPUT_INTERACTION that represent the entry and the exit points of a component, respectively. The behavior is defined through behavioral equations contained by the BEHAVIOR tag. A behavioral equation describes an internal path that can be run during the execution. Æmilia supports three port types for connecting components: (i) UNI that describes unidirectional ports, e.g., an input interaction that has one only input connection; (ii) OR that enables alternative paths starting or ending to the port; (iii) AND that enables not mutually exclusive paths on that port. Alongside the constant and architectural type section, the architecture topology contains: (i) the ARCHI_ELEM_INSTANCES tag that defines the component instances; ii) the ARCHI_INTERACTIONS tag that lists the connections between an entry point of a component with the exit point of another component; (iii) the ARCHI_ATTACHMENTS tag, which is a special kind of entry or exit point that is not linked to any OUTPUT/INPUT_INTERACTION, and where external requests can enter/exit the architecture.

Æmilia is equipped with an internal solver, namely TwoTowers, that enables different types of functional checks and performance analyses on a software architecture [37]. Two types of performance metrics have to be defined (in a specific reward file): (i) utilization through the STATE_REWARD tag, and (ii) throughput through the TRANS_REWARD tag. Listing 2 reports an excerpt of a reward file where, for example, the throughput of WSN.send_data a OUTPUT_INTERACTION is defined at lines 1 and 2.

¹ https://www.omg.org/spec/MARTE/About-MARTE/
The algorithm randomly generates the initial population $P_t$, shuffles it and applies the: (i) Crossover operator with probability $P_{\text{crossover}}$, and (ii) Mutation with probability $P_{\text{mutation}}$ that generate the $Q_t$ offspring. Thus, the obtained $R_t$ mating pool (by a doubled size) is sorted by the Non-dominated sorting operator, which lists Pareto frontiers (e.g., $F_i$) with respect to considered objectives. Finally, a Crowding distance is computed and a new family (i.e., $P_{t+1}$) is provided to the next step by cutting the worse half off.

Algorithm 1 The NSGA – II algorithm, reported from [38]

Step 1: Combine parent and offspring populations and create $R_t = P_t \cup Q_t$. Perform a non-dominated sorting to $R_t$ and identify different fronts: $F_i, i = 1, 2, ...$

Step 2: Set new population $P_{t+1} = \emptyset$. Set a counter $i = 1$. Until $|P_{t+1}| + |F_i| < N$, perform $P_{t+1} = P_t \cup F_i$ and $i = i + 1$. Step 3: Perform the Crowding-sort($F_i < c$) procedure and include the most widely spread ($N - |P_{t+1}|$) solutions to $P_{t+1}$ by using the crowding distance values in the sorted $F_i$.

Crowding-sort($F_i < c$)

Step 3.1: Call the number of solutions in $F$ as $l = |F|$. For each $i$ in the set, first assign crowding distance, $d_i = 0$

Step 3.2: For each objective function $m = 1, 2, ..., M$, sort the set in worse order of $f_m$ or, find the sorted indices vector: $I^m = \text{sort}(f_m)$

Step 3.3: For $m = 1, 2, ..., M$, assign a large distance to the boundary solutions, or $d_{i}^m = d_{max}^m = \infty$, and for all other solutions $j = 2, ... (l - 1)$, assign:

$$d_{i}^m = d_{max}^m + \frac{(f_{I^m_{i}} - f_{I^m_{j-1}})}{(f_{max}^m - f_{min}^m)}$$

Step 4: Create offspring population $Q_{t+1}$ from $P_{t+1}$ by using the crowded tournament selection, crossover and mutation operators.

SPEA2. Strength Pareto Evolutionary Algorithm (SPEA) has been introduced by Zitzler et al. [14], and Algorithm 2 describes its steps. Differently from NSGA – II, SPEA2 does not employ a non-dominated sorting process to generate Pareto frontiers. Therefore, SPEA2 needs additional data to create a Pareto frontier, as follows: (i) the population size $N$, the archive size $N$ in which non-dominated solutions are stored, and the maximum number of generations $T$, which is one of the stopping criteria.

SPEA2 randomly generates an initial population $P_t$ and an empty archive $A$, in which non-dominated individuals are copied at each iteration (see step 1 of the Algorithm 2).

For each iteration $t = 0, 1, ..., $ the fitness function values of individuals in $P_t$ and $A$ are calculated (see step 2). Then non-dominated individuals of $P_t$ and $A$ are copied to $P_{t+1}$, i.e., the next generation archive, by discarding dominated individuals or duplicates (with respect to the objective values). In case size of $P_{t+1}$ exceeds $N$, the truncation operator drops exceeded individuals by preserving the characteristics of the frontier. In case size of $P_{t+1}$ is less than $N$, dominated individuals from $P_t$ and $A$ are used to fill $P_{t+1}$ (see step 3).

When $t > T$, SPEA2 generates the non-dominated set $A$ in output, and the algorithm ends (see step 4). Until the stopping criterion is not met, SPEA2 generates the new population through operators (i.e., selection, crossover, and mutation), and increments the generation counter $t = t + 1$ (see steps 5, and 6).

3.3. Refactoring feasibility

Refactoring is a continuous activity during the software development, because developers change the source code structure due to maintenance issues while preserving the internal behavior. Therefore, any automated process should check the feasibility of refactoring actions to preserve the behavior. In case of a single action, it is quite easy to guarantee the refactoring correctness, while problems arise when multiple refactoring actions are applied in a row, and the sequence correctness has to be verified.

The feasibility of a refactoring sequence can be verified in two different ways: (i) post-execution, first the refactoring sequence is applied, and then its outcome is verified; or (ii) pre-evaluation of the sequence
feasibility, by checking in advance whether a sequence is feasible or not.

To the best of our knowledge, Opkyde showed in his Ph.D. thesis [39] the first framework for the pre-evaluation case within the object-oriented paradigm. First, each refactoring has to be equipped with a pre and a postcondition, then the framework tests the feasibility by checking the system’s status before and after applying the refactoring. Later, Cinnéide and Nixon have presented an extension to the Opkyde’s framework within the Java context [40]. In particular, their approach checks whether a precondition of an action in the sequence violates the postcondition of its previous action in the sequence. Hence, the approach is able to reduce single (pre)postconditions into global ones. In particular: (i) the global precondition is made up of the first action’s precondition in the sequence, plus each part of preconditions that are not yet verified by their previous action postconditions; (ii) the global postcondition is basically made up of all postconditions within the sequence.

Eq. (1a) shows an example of the aforementioned process, where $A_r$ is a refactoring action, and $P_r$ and $P_i$ represent the pre- and postcondition, respectively. In particular, the right part of Eq. (1a) is the final refactoring action derived from $A_r$ followed by $A_i$, Eqs. (1b) and (1c) describe how the Cinnéide and Nixon’s framework derives the global pre- and postcondition, respectively. The global precondition of the refactoring sequence shall be equal to: the precondition of the first action (i.e., $P_r A_r$) joint with the remaining part (i.e., / is the operator that removes the postcondition from a precondition) of the next preconditions that are not implicitly verified by the previous action postcondition (i.e., second part of the right-hand side of Eq. (1b)).

Hence, refactoring feasibility can be summarized as follows:

$$\begin{align*}
P_r A_r + P_i A_i & \longrightarrow P_r A_r^+ + P_i A_i^- \\
P_r A & = P_r A_r \land A_i^- / A_r^+ P_i^- \\
A_r^+ & = A_r^0 \land A_r^-.
\end{align*}$$

The global postcondition is trivially derived by joining all postconditions of actions belonging to the refactoring sequence, as defined in Eq. (1c).

4. EASIER

4.1. Original EASIER

Fig. 2 illustrates the EASIER framework. EASIER is an architecture-based evolutionary tool that employs a multi-objective genetic algorithm to discover Pareto frontiers. EASIER was presented in [12], and its core is grounded on JMetal [41]. JMetal is an open-source Java-based library that offers different evolutionary algorithm implementations, such as the NSGA – II [38] that is exploited by EASIER.

EASIER takes in input an initial software architecture and searches the architectural space (i.e., the solution space) by (re-)combining refactoring actions extracted from a repository, i.e., the Refactoring Actions Library in Fig. 2. The search process is driven by three objectives, namely the performance quality indicator ($perfQ$), the architectural distance ($archDist$) that is a measure of the intensity of changes induced by refactoring actions, and the number of performance antipatterns ($#PAs$) occurring in the software architecture $Æmilia$ [42].

We assign to each action a specific weight, which represents the effort to reproduce the action into the system. Therefore, heavier weights are assigned to actions, such as component cloning, that are considered more onerous than other ones. The combination of action weights within a solution determines the value of $archDist$ that has to be minimized.

EASIER produces sequences of refactoring actions, as output, in the form of a Pareto frontier, according to its evolutionary paradigm. As a consequence, the Pareto frontier can be used to obtain near-optimal architectural alternatives generated by applying sequences of refactoring actions to the initial architecture.

Fig. 3 depicts the EASIER flow. The process can start either with the textual representation of the input architecture (along with the reward specification), or with its Ecore representation (i.e., the $Æmilia$ model conforming to the $Æmilia$ metamodel). Since both representations are necessary for sake of EASIER work, two transformations (M2T and T2M in the figure) are provided to generate one representation from the other one.

EASIER invokes the native TwoTowers performance solver on the textual representation, which calculates the performance indices defined in the reward file. Thereafter, EASIER fills back the values of indices into the model. Once filled the model, the performance antipattern detection starts and identifies performance flaws. Thereafter, EASIER applies refactoring actions on the model, thus a refactored textual representation can be re-generated.

4.2. Extended EASIER

In this section, we summarize the extensions that we introduce in this paper to the EASIER approach, which are mostly aimed at overcoming threats to validity arisen in our previous work [12]. We have partitioned them in three areas, that are: experimental settings; refactoring actions and Pareto solutions; performance antipattern detection.

Experimental settings.

1. Extended EASIER can support an additional evolutionary algorithm, beside NSGA – II, that is SPEA2.
2. Original EASIER was observed to spend most of its execution time on performance analyses, thus we have here introduced multi-thread executions.
3. Extended EASIER is equipped with the JCommander command-line interface, which allows to manipulate the configuration file where the user can set different configuration characteristics, such as the action weights. Listing 3 reports a configuration file example. Lines 1–11 are related to the evolutionary algorithm characteristics, where Lines 2–5 set the stopping criterion, Lines 6–11 set the number

2 http://jcommander.org/
Fig. 2. The EASIER framework.

Fig. 3. The EASIER refactoring flow.

of independent runs for each algorithm, the used algorithms, and the quality indicators that will be used to analyze the Pareto frontier goodness. From Line 12 to 24 there are settings related to the problem itself. The user can specify the Æmilia solver’s path (Lines 12–14), and she can also set characteristics related to refactoring actions and how they shall be treated during the process. For example, the user can change the weight of a refactoring action type, and this might induce different selections during the solution process. Last lines (i.e., from 25 to 37) instead, set running configurations, such as the output location and the input model path.

4. In order to improve usability, extended EASIER only needs either an Æmilia initial textual architecture or an Æmilia model, because it autonomously generates the other needed files.

Refactoring actions and Pareto solutions.

1. In original EASIER, we had assumed predefined and fixed weights for the contributions of refactoring actions to the ArchDist objective [12]. In extended EASIER we enable the user to specify different weights in the configuration file.

2. Our results with original EASIER have suggested that diversity within a Pareto frontier is preserved and, in addition, the longer $\#evaIs$ increases, the more the performance quality of Pareto solutions increases. Hence, in extended EASIER, we have introduced appropriate indicators aimed at showing the quality of generated Pareto frontiers.

3. We mitigate the randomness factor of genetic algorithms by introducing the concept of independent runs that reduces this factor and, thus, improves the quality of the solutions.

4. Another powerful EASIER’s extension is the generation of charts and tables. In particular, extended EASIER exploits the JMetal’s plotting engine, which is able to create $\LaTeX$ tables and $R$ scripts,3 once the results are available. We use this extension in Section 6, where results are analyzed.

5. We equip EASIER with a new refactoring action, named RemoveClonedRefactoringAction, which is able to detect and remove a replica of an AEI within an Æmilia model. This action turned to be particularly useful for answering RQ3, namely for showing the independence of

3 https://www.r-project.org/
Listing 3: An excerpt of EASIERS’s configuration parameters

An excerpt of EASIERS’s configuration parameters is as follows:

```
#settings related to the JMetal
--maxEvaluation 2560
--populationSize 40
--independent_runs 31
--algorithm NSGA-II, SPEA2
--quality_indicator SPREAD, IGD+, IGD, EPSILON, HYPER_VOLUME, GENERALIZED_SPREAD

#AEmilia solver
--ttKernel easier-twoTowersSolver/bin/TTKernel
--length 4
--maxCloning 3
--cloningWeight 1.5
--constChangesWeight 1
-factor_of_change 0.5

#settings related to the problem
-outF target/output_folder
--temporary_folder target/tmp
--cleaning_temporary_file true
--input_model models/FTA
--oclTemplate detectionSingleValuePA.ocl
--generateReferenceFront true
```

5. Case study

A Fire Tracker Alarm System (FTAS) is a distributed system equipped with sensors and actuators that collaborate to discover a fire risk inside a building effectively. The system shall also drive the rescue teams by sensing the environment. For example, messages about the number of persons inside the fired building, the status of different fired edifice areas can be used by the rescue teams to understand and manage the most critical situations.

Fig. 4 shows a graphical representation of the software architecture of an FTAS. Each rectangle represents an architectural element type (i.e., a software component), following the AEmilia syntax, whereas each square on the boundary of a component represents an AEmilia input/output interaction. We have also used a colored representation for different types of interactions. The orange square is the external interaction of the system, that is where the workload insists; grey squares are passive interaction, i.e., a port without execution rate that is invoked by an active interaction; green squares, instead, represent output ports for active interactions, which are equipped with execution rates. Therefore, on the green ports, we can measure performance attributes, such as throughput. A filled black arrow in Fig. 4 represents a connection between an output interaction and the relative input one through the execution flow. Instead, a dashed arrow, i.e., an arrow contained in an architectural element type, is part of the internal execution flow. In addition, numbers over the dashed arrows indicate the sequence in the flow.

Fig. 4 represents the scenario where data gathered by a wireless sensor network (i.e., WSN) are collected by the fire tracker alarm component (i.e., FTA) by reading over a communication channel (i.e., CHN). Once data are available, they are sent through a LAN connection

5. Case study

A Fire Tracker Alarm System (FTAS) is a distributed system equipped with sensors and actuators that collaborate to discover a fire risk inside a building effectively. The system shall also drive the rescue teams by sensing the environment. For example, messages about the number of persons inside the fired building, the status of different fired edifice areas can be used by the rescue teams to understand and manage the most critical situations.

Fig. 4 shows a graphical representation of the software architecture of an FTAS. Each rectangle represents an architectural element type (i.e., a software component), following the AEmilia syntax, whereas each square on the boundary of a component represents an AEmilia input/output interaction. We have also used a colored representation for different types of interactions. The orange square is the external interaction of the system, that is where the workload insists; grey squares are passive interaction, i.e., a port without execution rate that is invoked by an active interaction; green squares, instead, represent output ports for active interactions, which are equipped with execution rates. Therefore, on the green ports, we can measure performance attributes, such as throughput. A filled black arrow in Fig. 4 represents a connection between an output interaction and the relative input one through the execution flow. Instead, a dashed arrow, i.e., an arrow contained in an architectural element type, is part of the internal execution flow. In addition, numbers over the dashed arrows indicate the sequence in the flow.

Fig. 4 represents the scenario where data gathered by a wireless sensor network (i.e., WSN) are collected by the fire tracker alarm component (i.e., FTA) by reading over a communication channel (i.e., CHN). Once data are available, they are sent through a LAN connection

Optimal solutions from the search starting point in the solution space.

Performance Antipattern detection.

Threshold calibration is a crucial task for performance antipattern detection. Since the number of performance antipattern occurrences is used as an objective of the fitness function, and our performance analysis is grounded on them, we have extended EASIERS by introducing the thresholds updating. We have implemented an engine for calculating and updating thresholds for each refactored version of the initial architecture. In particular, once a refactored AEmilia model is solved (i.e., performance indices are obtained), the engine updates all threshold values with respect to the current values of the refactored model. Thus, updated thresholds shall improve the quality of the Pareto frontiers solutions by enabling a more realistic and trustworthy performance antipatterns detection step.

Table 1 shows the configuration characteristics.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Eligible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome’s length</td>
<td>2, 4, 6</td>
</tr>
<tr>
<td>Cloning action weight</td>
<td>1.5, 4</td>
</tr>
<tr>
<td>Max number of cloning</td>
<td>3, genome length</td>
</tr>
<tr>
<td>Used algorithms</td>
<td>NSGA-II, SPEA2</td>
</tr>
</tbody>
</table>

Fig. 5. AEI cloning percentage within all the generated Pareto frontiers.
data, DSK shows data to the user, and SCR is responsible of encrypting (i.e., LAN) to: database (i.e., DB), desktop application (i.e., DSK), and

\[ \mu \]

Table 3

<table>
<thead>
<tr>
<th>Configuration</th>
<th>#clones</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL2,WCA1.5,MC3</td>
<td>0%</td>
<td>13%</td>
</tr>
<tr>
<td>GL2,WCA4,MC3</td>
<td>0%</td>
<td>13%</td>
</tr>
<tr>
<td>GL4,WCA1.5,MC3</td>
<td>34%</td>
<td>31%</td>
</tr>
<tr>
<td>GL4,WCA4,MC3</td>
<td>37%</td>
<td>31%</td>
</tr>
<tr>
<td>GL6,WCA1.5,MC3</td>
<td>22%</td>
<td>21%</td>
</tr>
<tr>
<td>GL6,WCA4,MC3</td>
<td>23%</td>
<td>22%</td>
</tr>
<tr>
<td>GL6,WCA4,MC4</td>
<td>38%</td>
<td>34%</td>
</tr>
<tr>
<td>GL6,WCA4,MC5</td>
<td>38%</td>
<td>35%</td>
</tr>
<tr>
<td>GL6,WCA4,MC6</td>
<td>24%</td>
<td>20%</td>
</tr>
</tbody>
</table>

Table 4

<table>
<thead>
<tr>
<th>Strength</th>
<th>Pearson's index</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>1 to 3</td>
<td>.17</td>
</tr>
<tr>
<td>Medium</td>
<td>.3 to .5</td>
<td>.35</td>
</tr>
<tr>
<td>Large</td>
<td>.5 to 1.0</td>
<td>.50</td>
</tr>
</tbody>
</table>

(i.e., LAN) to: database (i.e., DB), desktop application (i.e., DSK), and security component (i.e., SCR). In fact, DB is in charge of storing the data, DSK shows data to the user, and SCR is responsible of encrypting and decrypting data.

Table 2

<table>
<thead>
<tr>
<th>Run no.</th>
<th>evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>640</td>
<td>1280</td>
</tr>
<tr>
<td>1920</td>
<td>2560</td>
</tr>
<tr>
<td>3200</td>
<td>3840</td>
</tr>
<tr>
<td>4480</td>
<td></td>
</tr>
</tbody>
</table>

6. Experimental analysis

With respect to our previous work [12], we have introduced here the possibility of varying several internal configuration characteristics in order to investigate their correlation with the quality of the results. Table 1 reports all the eligible values for each considered characteristic.

First, the user can set a different genome's length at her convenience. In our experimentation, we have set this length to different values to investigate its correlation with performance improvement. We have also considered that an extremely long genome would not be a useful solution, since it generates a refactored architecture very distant from the pareto frontier across different configurations.

This is the scenario that we have considered for our experimentation. However, Fig. 4 also contains dashed arrows and components, as example results of refactoring actions, as well as new bold rates of actions, which will be illustrated later.
Fig. 6. Percentage of constant change refactoring actions within NSGA-II and SPEA2 Pareto frontiers.

(a) NSGA-II

(b) SPEA2

Fig. 7. Sensitivity of solutions to FOC for the NSGA-II algorithm with Genome length 2, while varying the cloning action weight and its maximum number. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)
from the initial one. Hence, we have limited the genome length to six individuals.

Second, in extended EASIER, it is possible to set the weight of a cloning action. This characteristic is especially relevant in combination with the reference unit of weight. We recall that these weights determine the archDist objective value, hence the ratio between the two refactoring action weights may favor the occurrence of actions in the solutions found. For this reason, in our experimentation, we have set the cloning action weight to two quite different values (i.e., 1.5 and 4).

Again on the cloning action, the user can now limit the maximum number of cloning for each genome. In our experimentation, we have set once the limit to three (as in the original EASIER), and then to the genome’s length. In the original EASIER, we had limited this number due to computation time, because cloning actions produce larger Æmilia models that need more time to be processed. Since we have now mitigated this issue by introducing multi-thread performance analysis, it is feasible to consider a larger number of cloning actions up to the genome length.

Finally, extended EASIER supports two different evolutionary algorithms, namely NSGA – II and SPEA2. We compare the SPEA2 near-Pareto frontiers against NSGA – II ones to analyze the quality of obtained solutions.

In order to validate our approach, we have created a benchmark of twelve configurations by varying, for each used algorithm, the characteristics reported in Table 1.
have changed the maximum number of epochs (i.e., #epochs) from 10 to 70 (by a step of 10). We remark that #epochs represents the epochs during which a genetic algorithm combines individuals to generate the new offspring. EASIER calculates the maximum number of evolutions (i.e., a stopping criteria) as #evals = pop * #epochs. Hence, by varying #epochs, we can control the Pareto frontiers quality, since the longer is the evaluation, the better is the frontier.

6.1.1. \( RQ_1.1 \): role of cloning

Table 2 reports the IGD+ indicator values, where different runs are on the rows and the #evals values obtained while varying #epochs as described above are on the columns. While moving from left to right along columns, the runs require longer execution times due to higher #evals values. By observing the AVG row at the bottom of the table, we can notice that the minimum average IGD+ value is around 0.16, which is obtained for different values of #evals. Hence, due to the above execution time consideration, we have selected the leftmost column in which it is achieved, that is the bold one. Therefore, #evals = 2560 has been adopted in our experimentation.

6.2. Results discussion

In the following, we discuss in detail the results in terms of correlation between refactoring actions and performance variation. In particular, we are interested in the influence of configuration characteristics on the goodness of Pareto frontiers’ solutions. From now on, we use the following notation to identify a specific configuration: GLx_WCAy_MCz, where x expresses the genome’s length (GL), y expresses the weight assigned to a cloning action (WCA), and z expresses the maximum number of clones (MC) within a genome.

As defined in the introduction, we recall that our experimentation is aimed at answering the following research questions:

- \( RQ_1 \): Which evolutionary algorithm proposes more interesting near-Pareto solutions on our multi-objective optimization problem between \( NSGA-II \) and SPÆA2, while varying their configuration characteristics? In particular:
  - \( RQ_1.1 \): What is the role of the cloning action within obtained solutions?
  - \( RQ_1.2 \): What is the role of change rate action within obtained solutions?

- \( RQ_2 \): Which evolutionary algorithm better inspects the solution space on our multi-objective optimization problem between \( NSGA-II \) and SPÆA2, while varying the configuration characteristics?

- \( RQ_3 \): Is EASIER able to reach a desired solution independently of the search starting point within the solution space?

First, we have conducted a study on the percentage of cloning within near-Pareto frontiers. Figs. 5(a) and 5(b) show two pie-charts with the AEI cloning percentage within all the generated Pareto frontiers by \( NSGA-II \) and SPÆA2, respectively.

\( A \) replication package of our experiments can be found here: https://git.io/easier-rep-pkg.
As a first analysis, SCR and FTA are cloned with the same percentage both in case of NSGA – II and SPEA2. Therefore, the software architecture shall contain at least a replica of the FTA and the SCR to show either better performance or reduce the number of detected performance antipatterns. Furthermore, it is also clear that the DSK and LAN AEIs do not require to be cloned in near-Pareto frontiers (i.e., both were never cloned). Moreover, the NSGA – II algorithm never clones the CHN and WSN AEIs, while it creates one replica of the DB AEI out of five clones (i.e., 20%). The SPEA2 near-optimal Pareto frontiers, instead, are a bit different. In fact, DB is the third most cloned AEI, in 15% of the cases, whereas WSN and CHN have been cloned only in 2% of the cases.

Both the FTA and LAN AEIs are central in the architecture in Fig. 4, thus it may be expected that both AEIs are subject to cloning actions. As opposite, only FTA needs to be cloned in practice, for the following reason. In EASIER, we have introduced a specific Factor of change (FOC) applied to action rates within an Æmilia architecture, and we have bounded it to [0.5, 2]. In our example, the FOC range might not be large enough to include FTA rates that induce better performance, thus forcing EASIER to create a replica of the FTA AEI. Instead, EASIER can achieve better performance solutions by changing the LAN rates, thus it is not necessary to create a LAN replica.

We have then analyzed the percentage of cloning actions overall solutions, while varying configurations. In Tables 3a, 3b the first column reports the configuration (e.g., GL2_WCA1.5_MC3), whereas the second and third columns report the percentage obtained with NSGA – II and SPEA2 algorithms, respectively. Tables differ for the setting of maximum number of clones.

We have noticed that the SPEA2 algorithm considers more cloning actions within near-Pareto solutions where GL is equal to 2, due to its Pareto frontiers nature. In fact, by construction SPEA2 generates a wider frontier than NSGA – II, and this may result in worse solutions, i.e., a SPEA2 solution may contain dominated solutions (as mentioned in Section 3.2).

We can also notice that medium length genomes (i.e., GL4) correspond to the cases where the cloning actions have higher percentage of occurrences in both the algorithms (roughly 30%), and in both Tables 3a, 3b. By analyzing more in-depth this result, we have found that it is due to a limitation of the Æmilia grammar combined with the example architecture structure that we have considered. Indeed, the Æmilia grammar does not allow to have a connection between two ports (i.e., outgoing and incoming ports) with OR-OR nature. Our cloning action has been implemented in the way that it preserves the functional behavior, and this is achieved by changing the nature of the neighborhood ports of AEIs that come before and follow the cloned AEI in the architecture. In particular, it changes those ports to OR to distribute the requests among the original and cloned AEIs. Thus, the system behavior is preserved because the action does not change the original flow and, at the same time, the performance shall improve because the clone action can reduce the utilization of the target AEI. As a side effect, it is practically forbidden to clone two AEIs that are adjacent in the execution flow, and this is limiting in our case study.

Thereafter, we have analyzed the correlation among the performance improvement represented by the perfQ objective and the number of cloning actions (i.e., #clones) in near-Pareto frontiers, by using the Pearson’s correlation index. Table 4 relates the correlation strength to ranges of the index absolute value. If the Pearson’s index is near ±1, then it is said to be a perfect correlation: as one variable increases, the other one tends to also increase (if positive) or decrease (if negative).
6.2.1. \( RQ_1 \): role of action rates

Figs. 6(a) and 6(b) show the percentage of change in refactoring actions within NSGA – II and SPEA2 Pareto frontiers, respectively. In general, the most refactored constant in our example architecture is \( \text{dec\_rate} \) (by around 29%), which represents the rate of the decryption action. The second most changed constant is \( \text{lan\_sec\_rate} \), by around 20% in case of NSGA – II, and around 15% in case of SPEA2.

Since a constant can be changed by a Factor Of Change (FOC) that can be either an increment or decrease, as we said before, we have studied the sensitivity of solutions to FOC. Figs. 7 through 12 show 24 charts, where the first twelve ones are related to NSGA – II algorithm, while the second twelve ones are related to SPEA2. For each algorithm the charts are partitioned by genome length (i.e., GL), while varying the other configuration characteristic (i.e., WCA, and MC). The \( y \)-axis of each chart lists \( \text{Amilia} \) constants rates, while the \( x \)-axis reports the number of times FOC assumes a positive (i.e., red bar) or negative (i.e., blue bar) value within near-Pareto solutions.

It is worth noticing that \( \text{dec\_rate} \) and \( \text{lan\_sec\_rate} \), which are the most considered constants (as said above), are always changed by a positive FOC (e.g., \( \text{dec\_rate} \) in GL4_WCA4_MC4 has been positively changed 1019 times in case of SPEA2 and 101 times in case of NSGA – II). Hence, we can deduce that the architecture tends to go towards near-optimal solutions when both \( \text{dec\_rate} \) and \( \text{lan\_sec\_rate} \) are increased.

6.2.2. \( RQ_2 \): solution space inspection

In order to understand and compare Pareto frontiers, quality indices have been introduced [45,46]. A very used one is the Hyper-Volume (HV) that indicates how much of the solution space is covered by the algorithm. Unfortunately, HV is costly to calculate. Therefore, other quality indicators have been introduced, such as the Enhanced Inverted Generational Distance plus (IGD+), which calculates the distance between a solution and the optimal one. It is intuitive to deduce that the latter index should be as close to zero as possible, while the former one should be as much large as possible. Figs. 13, and 14 show box and whisker plots of HV and IGD+, respectively.
V. Cortellessa and D. Di Pompeo

In terms of HV, NSGA-II outperforms SPEA2 in all solutions except the case of GL_6_WCA4_MC6 in which SPEA2 better inspects the solution space. The latter case may be due to the structure of the Pareto frontier. In fact, as said above, SPEA2 produces larger frontiers than NSGA-II does, in which dominated solutions may be considered. Hence, in case of GL_6_WCA4_MC6 the larger Pareto is very likely made of better solutions.

With respect to IGD+, NSGA-II always outperforms SPEA2. In particular, by considering the GL_6_WCA4_MC6 problem where SPEA2 outperforms NSGA-II in terms of HV, in terms of IGD+ NSGA-II works better than SPEA2. In this case, perhaps a narrower frontier is helpful to obtain better performance.

By summarizing, based on our results, we can claim that NSGA-II produces better solutions against the SPEA2 ones. In particular, we have compared near-Pareto frontiers, obtained under different configurations, using well-known quality indicators, such as Hyper-Volume (HV) and Enhanced Inverted Generational Distance (IGD+). NSGA-II outperforms the SPEA2 in terms of HV in almost the cases. Hence, NSGA-II more widely covers the solution space volume than SPEA2 (see Fig. 13). In terms of IGD+, only in one case SPEA2 generates a comparable near-Pareto frontier (see Fig. 14).

6.2.3. RQ3: Is EASIER able to reach a desired solution independently of the search starting point within the solution space?

We answer RQ3 by introducing a process that generates worse architectures than a reference architecture and, by applying EASIER to the former ones, we check whether the process leads to optimal solutions that are close to the reference architecture, where the closeness criterion is defined in the following.

Hence, in order to obtain worse architectures, we have inverted the EASIER fitness function. In particular, we have asked EASIER to find architectures that minimize the PerfQ objective. We remind that PerfQ represents the performance variation indicator, and it can assume positive or negative values, thus its minimization leads towards solutions with negative values that represent performance degradation. We have run EASIER with the inverted fitness function and then we have ordered all generated solutions from the worst to the best one in terms of PerfQ values. We have then considered the first five solutions as different architectures that users might have designed, and we have appointed them as inputs to the optimization process. Thereafter, we have executed EASIER, starting from each of these five architectures, with the original fitness function (i.e., the one that maximizes PerfQ and minimizes the other two objectives) to obtain Pareto frontiers of non-dominated solutions. Finally, we have estimated the closeness of solutions in Pareto frontiers with the reference architecture, in order to evaluate the ability of EASIER to reach a desired solution starting from different points of the solution space.

In our context, we consider an architecture close to another one when their performance are similar. Therefore, we use PerfQ itself to quantify this closeness. The lower is the PerfQ absolute value, the closer are the considered architectures.

In Fig. 15 we have drawn five radar charts, where each chart depicts a Pareto frontier obtained from applying EASIER to one of the five worsened architectures considered as initial ones. In particular: Charts 15(a) and 15(b) depict Pareto frontiers obtained from initial solutions that both have LAN and CHN AEI cloned, and whose closeness to the reference architecture is around 35%; Charts 15(c), and 15(d) depict Pareto frontiers obtained from initial solutions that have only LAN AEI cloned, and whose closeness to the reference architecture is...
around 33%; Chart 15(c), instead, depicts the Pareto frontier obtained from an initial solution with LAN and WSN AEI cloned, and whose closeness to the reference architecture is around 34%.

Each triangle within a radar describes a solution belonging to that Pareto frontier. The radar axes are: (i) the number of changes; (ii) the number of performance antipatterns; (iii) the performance quality (i.e., $\text{PerfQ}$). For each solution, we have drawn the triangle having as vertices the three objectives, whose values are reported in the square boxes coupled to each radar in the figure. A special attention has to be given to $\text{PerfQ}$ that, as mentioned above, has been computed as the performance quality difference between a solution within the Pareto frontier (i.e., the triangle) and the reference architecture. In this way, the absolute value of $\text{PerfQ}$ represents the closeness between them.

Each radar caption reports the closeness of a whole Pareto frontier to the reference architecture, defined as the average closeness overall solutions belonging to the frontier. We observe that the average closeness in all five cases falls within 3% and 5%, while (as mentioned above) the closeness of initial architectures were falling between 33% and 35%. This indicates that EASIER is able to find acceptable solutions, close enough to the reference one, while starting from quite different points of the search solution space.

On the basis of our results, and considering the evolutionary optimization randomness, we can summarize that EASIER can suggest to the user near-optimal solutions with improved performance independently from the initial architecture.

6.3. Scalability

In order to evaluate the scalability of EASIER, we have measured the execution time needed to obtain near-Pareto frontiers while varying the configuration characteristics of NSGA-II and SPEA2. For sake of discussion, we report in Fig. 16 the execution time of three configurations for both the algorithms while varying the number of evolutions. The execution time is expressed in millisecond on the $y$-axis, and the number of evolutions is on the $x$-axis.

Notably, we have selected configurations with the maximum number of clones equal to the genome length since they more likely generate solutions equipped with a large number of clones. Indeed, we have empirically found that most of the time is taken by the Æmilia solver (i.e., TTKernel) to obtain performance indices. Therefore, solutions with a high number of clones take more time since the Æmilia model has to be transformed into a Markov chain made of a high number of states.

By analyzing these three configurations, we can note that SPEA2 requires 5.55 h to compute a Pareto frontier with 4480 evals, while NSGA-II requires 3.33 h to solve the same configuration. Thus, in this case NSGA-II outperforms SPEA2 in terms of execution time, as reported also in some previous work [47].

In our tuning phase (see Section 6.1), by considering the execution time needed to obtain a Pareto frontier with the highest number of evolution (i.e., 4480) and the quality indicator (i.e., $\text{IGD}^+$) as shown...
in Table 2, we have decided to use 2560 as the number of evolution, which generates Pareto frontier with $\text{IGD}^+ = 0.163$. Moreover, we have also decided to use a genome length 4, which requires to $\text{NSGA-II}$ 1.388 h on average for each Pareto frontier, because it represents a reasonable number of refactoring actions to be applied in a row.

7. Threats to validity

In this section, we discuss threats to validity of our approach by following their classification in [48].

**Conclusion validity.** We have executed thorough experiments, which have produced a lot of data on the specific architecture example that we have considered. Thus, we are not affected by Low statistical power, and Violated assumptions of statistical tests. Meanwhile, we might be affected by the Fishing and the error rate threat, namely we might be blinded in searching a specific result. This is not a real problem in our approach, because it is expected to produce multiple suggestions to software architects for performance improvements, hence it is not aimed at finding any specific solution.

In order to remove, instead, threats related to measurements and implementation, such as Reliability of measures and Reliability of treatment implementation, we have executed thirty-one different runs for each problem. The idea is to remove the randomness of the evolutionary algorithms by repeating different runs. Hence, our approach shall be resilient with respect to these threats.

**Internal validity.** In order to mitigate selection threats, we have configured our experiments in the way that multiple executions are run. Indeed, a selection threat can be due to the way EASIER selects a solution among all the ones within the solution space. We mitigate this threat by introducing multiple repetitions for each configuration. This mechanism might also mitigate threats concerning the ambiguity about direction of causal influence. In fact, as we have reported in Table 5, we have encountered high correlations between the performance improvements and the number of clones within Pareto frontiers, and we rely that no casual influences have threatened our results.

**External validity.** Due to the nature of the modeling language that we have considered, it is clear that we cannot make deductions within different modeling contexts, which could be possibly adopted in real industrial environments. In particular, the portfolio of refactoring actions is affected by the considered modeling language because its syntax and semantics determine the amount and type of refactoring actions that can be put in place. Hence, the EASIER porting to other languages has to be investigated.
Fig. 15. Radar charts of Pareto solutions w.r.t. to reference architecture.

(a) Average closeness = 4.1%
(b) Average closeness = 3.1%
(c) Average closeness = 4.5%
(d) Average closeness = 4.6%
(e) Average closeness = 4.3%

Fig. 16. Execution times of a chunk of experiments for NSA II and SPEA2 over different evaluations. The x-axis represents the number of evaluations, while the y-axis represents the execution time expressed in millisecond.

(a) NSGA II
(b) SPEA2.
8. Conclusion

In this work, we have analyzed the sensitivity of multi-objective software architecture refactoring to the optimization algorithms configuration characteristics. In particular, we have investigated how the goodness of Pareto frontiers can be affected by varying both characteristics related to the searching algorithm, such as the length of the genome, and characteristics related to the problem, such as the composition of refactoring sequences.

From the analysis of our results, several interesting aspects have emerged. For example, we have shown that both the used algorithms have found that the initial architecture of our case study suffered of a few architectural elements within a specific flow path, thus solutions often contain replicas of those elements.

In our experimentation, we have also compared two different algorithms (i.e., NSGA-II and SPEA2). In particular, we have analyzed different quality attributes, which have highlighted that NSGA-II outperforms SPEA2 in most of cases. However, the SPEA2 algorithm generates more scattered solutions that the NSGA-II ones across the solution space. This could be a relevant insight in cases where solution diversity is important.

As future work, we intend to tackle the threats to validity discussed before. In particular, we intend to exploit this experience in model-based performance-driven refactoring in other modeling contexts, such as UML.


