EASIER: Evolutionary Approach for multi-objective Software architecture Refactoring

Davide Arcelli, Vittorio Cortellessa, Mattia D’Emidio and Daniele Di Pompeo
EASIER: an Evolutionary Approach for multi-objective Software architecture Refactoring

Davide Arcelli, Vittorio Cortellessa, Mattia D’Emidio, Daniele Di Pompeo
University of L’Aquila, Italy
{davide.arcelli, vittorio.cortellessa, mattia.demidio}@univaq.it
daniele.dipompeo@graduate.univaq.it

Abstract—Multi-objective optimization has demonstrated, in the last few years, to be an effective paradigm to tackle different architectural problems, such as service selection, composition and deployment. In particular, multi-objective approaches for searching architectural configurations that optimize quality properties (such as performance, reliability and cost) have been introduced in the last decade. However, a relevant amount of complexity is introduced in this context when performance are considered, often due to expensive iterative generation of performance models and interpretation of results. In this paper we introduce EASIER (Evolutionary Approach for multi-objective Software architecture Refactoring), that is an approach for optimizing architecture refactoring based on performance and on the intensity of changes. We focus on the actionable aspects of architectural optimization, instead of merely searching over a set of alternatives. We also start to investigate on the potential influence of performance antipatterns on such process. We have implemented our approach on Æmilia ADL, so to carry out performance analysis and architecture refactoring within the same environment. We demonstrate the effectiveness and applicability of our approach through its experimentation on a case study.

I. INTRODUCTION

Since more than one decade multi-objective optimization has been applied to several software architecture problems, and it has demonstrated to be a particularly effective paradigm on problems that can be natively formulated through quantifiable metrics. The optimization of quality attributes nicely fits into this category of problems, because these attributes (such as performance, reliability, usability) are meaningful only if expressed through well-defined metrics [1]. In fact, the ability of software engineers to satisfy quality requirements depends on the possibility of comparing metric values to these requirements.

With regard to architecture quality attributes, the evaluation of performance metrics is a particularly complex process, because they emerge from the combination of several software characteristics, i.e., static, dynamic, deployment and - in some cases - environmental ones. Beside this, very few ADLs embed constructs to specify performance parameters, and even fewer ones provide tools to natively analyze performance within the same ADL environment. 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 [2].

Several multi-objective approaches have been introduced, in the last decade, to optimize the performance of a software architecture along with other quality attributes, such as reliability and cost [3], [4]. Many of such approaches are based on evolutionary algorithms [5] that allow to search the solution space by (re-)combining solutions.

A common character of these approaches 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 aim at considering in this paper.

We here introduce EASIER, that is an Evolutionary Approach for Software architecture Refactoring, aimed at optimizing metrics related to the performance and to the distance from the initial architecture. EASIER deals with genomes that represent sequences of refactoring actions extracted from a repository, i.e., the Refactoring Actions Library in Fig. 1. The search process is driven by three main objectives codified in a fitness function, that are performance quality indicators, the architectural distance (that will be expressed...
as a measure of the intensity of changes induced by refactoring actions), and the number of performance antipatterns occurring in the software architecture.

Figure 1: The EASIER high-level architecture.

It is important to distinguish here between quality attributes that can be represented by analytical models (e.g., architectural cost) and the ones that claim for complex models to be solved in order to achieve an adequate accuracy with respect to architectural details (e.g., resource contention-based performance indices). In our case, an analytical model underlies the architectural distance, while a Performance Analyzer is devised (see the bottom of Fig. 1) to analyze the performance and detect performance antipatterns within the architecture.

EASIER produces sequences of refactoring actions that induce, as output, in the form of a Pareto frontier, according to its evolutionary paradigm. The latter is made of the architectural alternatives generated by those sequences that lead to non-dominated solutions.

The EASIER low-level architecture has been designed to apply our evolutionary algorithm to software architectures described in different ADLs. In Section III we specify the entry points left in EASIER for this goal, even though it is out of this paper scope to evaluate the effort needed for plugging new ADLs within EASIER. In this paper we have chosen Æmilia as our ADL context [8], mostly because the Æmilia context natively enables performance analysis within its own environment, without the need of generating performance models in different notations. We have also exploited an existing approach for performance antipattern detection in Æmilia [9].

The paper is organized as follows: Section II presents the related work, Section III illustrates our approach, Section IV shows its validation on a case study, and Section V concludes the paper.

II. RELATED WORK

With the continuous evolution of software systems even after release, automation in software refactoring has become a critical need along the whole development process [10]. In fact, many studies have been conducted in the context of model-based software refactoring (see, e.g. [11]–[13]).

However, finding the best sequence of refactoring actions to be applied to a software artifact, in order to optimize its quality w.r.t. to a set of metrics (a.k.a. objectives), is a problem known to be computationally hard, due to the typically huge space of feasible solutions [14]. 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 [15]–[18].

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 [8], [19]–[21]) and with different degrees of freedom to modify the architecture (e.g., service selection, composition or deployment [22], [23]. A systematic literature review on architecture optimization can be found in [1].

An interesting contribution in this direction was given in [3], [24], 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 [25] and produces, through model transformation, a Layered Queueing Network for sake of performance analysis. Instead, our approach works entirely within the Æmilia ADL environment, hence it is not subject to changes of notation that may induce inaccuracies into the performance model.

Another relevant approach has been introduced in [26], 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 suffers of 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...
An approach taking place in a unique environment for modeling and analysis has appeared in [4]. A tool is introduced, based on AADL [27], 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 unique 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.

III. THE EASIER ARCHITECTURE

In this section, we describe the low-level architecture of EASIER that is schematically illustrated in Fig. 2. The figure is vertically divided in two swimlanes. On the left, we report the evolutionary context of the approach, while on the right we report the ADL context. Fig. 2 is also horizontally divided in two major swimlanes, i.e. Data and Process. Data are, in turn, partitioned in: metadata on which architecture refactoring and algorithm solution are founded, and files and libraries (i.e. knowledge) directly feeding the process.

A. Evolutionary context

The bottom left side of Fig. 2 illustrates the EASIER core, that is a multi-objective evolutionary algorithm, namely customNSGAII. It essentially consists in a customization of the well-known NSGA-II algorithm [7], which has been developed to properly take into account the specific nature of the optimization problem we deal with. In particular, we adopted JMetal as a building block, which is a well established object-oriented Java-based framework for multi-objective optimization with metaheuristics [28] that comprises a basic implementation of the NSGA-II algorithm. The latter has been selected due to its wide adoption in the software engineering community [29], and to its extension capabilities. We have also been able to use some of the available features as they were, in particular the representation/storage of objectives and solutions, as well as their manipulation, and the selection operator [7].

Notwithstanding the reused JMetal features, the context of our optimization problem required to heavily customize parts of the baseline framework, by tailoring some interfaces exposed by the latter, as detailed in the following. Before describing the algorithm and the NSGA-II customizations introduced in EASIER, however, we describe the main data which customNSGAII relies on, shown in the top left side of Fig. 2

1) Data: customNSGAII exploits the concept of Solution, which contains the representation of a genome as a Refactoring that is a sequence of a number len of architectural RefactoringActions. Both Refactorings and RefactoringActions have PreConditions and PostConditions, which are first-order logical formulae evaluated during the evolutionary process to determine their feasibility. The adopted mechanism for calculating and verifying Refactorings and RefactoringActions pre- and post-conditions is an implementation of the one in [30], which essentially produces Refactoring conditions by elaborating the conditions of its RefactoringActions. To this aim, we have developed an approach based on: (i) an ad-hoc metamodel supporting the definition action pre- and post-conditions as logic formulae, and (ii) code generation facilities implementing the mechanism for calculating and verifying Refactoring conditions while composing RefactoringActions [10]. We do not provide details on this aspect because it is out of this paper scope.

It is worth to remark that Refactorings are intended to be “aggregators” of RefactoringActions, because the latter may exist independently from the former. RefactoringActions are stored into an ad-hoc repository named Refactoring Actions Library. A RefactoringAction represents one of the entry points of EASIER introduced to plug different ADL contexts, as it will be illustrated in Section III-A2.

A Solution also contains a reference to the corresponding alternative architecture resulting from the application of the genome sequence. Such alternative architectures are assumed to be conforming to a specific ADL. The application of each refactoring is necessary to analyze the performance of the generated architectural alternative, as it will be described in Section III-A2.

Each Solution has three attributes that together represent the objectives of our fitness function, namely ArchDist, PerfQ and #PAs.

To ease an ADL hosting, we introduce a configuration file to set all the required input parameters. Some of these are related to the evolutionary process, namely len, pop, evals, p(xover), and p(mut). In particular, len defines the genome length, pop (evals, resp.) determines the population (the number of epochs, resp.) used by of the evolutionary algorithm, whereas p(xover) and p(mut) represent the crossover and mutation probabilities, respectively, that affect the way the solution space is explored.

2) Process: Conformingly to the typical NSGA-II flow [7], the first iteration of the algorithm consists of a generation phase, aimed at randomly creating an initial
population of candidate solutions (i.e. refactorings by length) with a pop cardinality. This phase is a customized step where, every time customNSGAI needs to generate a new candidate solution, a feasible Refactoring is generated by incrementally concatenating compatible, randomly generated, RefactoringActions. In particular, two RefactoringActions $a_i$ and $a_{i+1}$ are said to be compatible if and only if the preconditions of $a_{i+1}$ are not violated by the postconditions of $a_i$. Accordingly, a Refactoring $R$ is said to be feasible if all its consecutive RefactoringActions are compatible.

After the (custom) generation of the initial population, solutions are evaluated according to a customized fitness function that, in EASIER, considers three objectives to optimize, namely ArchDist, PerfQ and #PAs, which are defined in the following.

PerfQ (to maximize). It represents a performance quality indicator aimed at quantifying the relative performance improvement induced by a refactoring w.r.t. an initial architecture, defined as follows. Let $I$ be the result of a
performance analysis on the initial architecture W.R.T. to a vector of $c$ performance measures of interest (e.g. throughputs or utilizations of components of the system), and let $I_k$ denote the $k$-th element of this vector (e.g. a throughput value). Moreover, let $F$ be the result of a performance analysis, W.R.T. the same measures, on a generic architectural alternative $A$, obtained by applying a refactoring to the initial architecture. Analogously, let $F_k$ denote the $k$-th element of $F$. Then, the performance quality indicator of $A$ is defined as $\text{PerfQ}(A) = \frac{1}{c} (\sum_{j=1}^{c} p \cdot \frac{(F_j - I_j)}{I_j + I_j})$, where $p \in \{ -1, 1 \}$ is a multiplying factor that holds: i) $1$ if the $j$-th measure has to be maximized (i.e., the higher the value, the better the performance), like the throughput; ii) $-1$ if the $j$-th measure has to be minimized (i.e., the smaller the value, the better the performance), like response time. In this way, a decrease (increase, resp.) in the value of a measure that one would like to minimize (maximize, resp.) is interpreted as a positive contribution (and viceversa). In fact, each $j$-th term of the sum within the computation of $\text{PerfQ}(A)$ will be: i) a positive real when we aim at minimizing (maximizing, resp.) the $j$-th measure and such a measure, in the architectural alternative, exhibits a value $F_j$ that is smaller (larger, resp.) than that assumed in the original architecture, i.e. $I_j$; ii) a negative or zero value otherwise. Hence, architectural alternatives whose overall performance is better (worse, resp.) than the initial one will be associated with positive (negative, resp.) values of performance quality indicator, as $\text{PerfQ}(A)$ will be a positive real only when the majority of the terms contribute with positive values. For this reason, we consider $\text{PerfQ}$ as part of our fitness function (to maximize), so that refactorings leading to architectural alternatives providing better performance are preferred over others. Notice that, for performance measures representing utilizations, $p$ also holds $1$ but, similarly to [20], we define a correction factor $\Delta_j$, to be added to each $j$-th term above, whose purpose is to penalize refactorings that push the utilization too close to its maximum value of $1$. In particular, our algorithm tends to maximize utilization up to a certain threshold, whereas utilizations higher than this threshold are rather considered as risky. For sake of our implementation, we adopt a threshold value of $0.8$, but this can be easily changed within EASIER. In particular, we define:

$$
\Delta_j = \begin{cases} 
\frac{\sqrt{I_j} - I_j}{2\sqrt{I_j} + I_j} & \text{if } F_j > 0.8 \land I_j > 0.8 \\
0.8 - F_j & \text{if } F_j > 0.8 \land I_j \leq 0.8 \\
I_j - 0.8 & \text{if } F_j \leq 0.8 \land I_j > 0.8 \\
0 & \text{otherwise}
\end{cases}
$$

ArchDist (to minimize). It quantifies the distance of an architectural alternative $A$ from the initial one, in terms of intensity of refactoring changes. The distance of $A$ from an initial one is defined as the sum of the distance induced by each RefactoringAction $a_i$ in the correspond-
non-dominated ones, namely the Pareto frontier, is returned.

Finally, we remark that each generated architectural alternative undergoes a Performance Analysis process in order to obtain performance indices of interest for the corresponding architectural model. Such process strictly depends on the target ADL, which may need some processing before and after plugging a specific performance analyzer within EASIER. In other words, a "bridge" between customNSGAII and the Performance Analyzer has to be provided that, in one direction, calls a solver for the architectural model and, in the other direction, properly fills back performance measures into the model.

B. Æmilia ADL context

On the right side of Fig. 2 we show how an ADL context can be plugged into EASIER. In this description we make specific reference to Æmilia, that is the ADL we have chosen for sake of this paper [8].

1) Data: In order to plug a new ADL in EASIER, ADL-specific refactoring actions have to be provided by software architects, as specializations of RefactoringAction, which is defined as an EASIER entry point in the evolutionary context. Hence, ADL-specific refactoring actions inherit pre- and post-condition attributes that have to be defined and implemented within the specific ADL context. As mentioned in Sec III-A1, we have developed an approach that eases actions implementation because it grounds on an ad-hoc metamodel and code generation facilities.

We envision the plugging mechanism to the RefactoringAction entry point as being implemented conformingly to the pattern reported in Fig. 2; a specialization of RefactoringAction is first created and then specialized as ADL-specific refactoring actions. In the Æmilia context, the RefactoringAction specialization is represented by the ÆmiliaRefactoringAction concept. For sake of this paper scope, we have developed an initial pool of predefined refactoring actions for the Æmilia ADL context, which is currently composed by two actions, namely CloneAEI and ChangeRate.

CloneAEI is in charge of cloning a srcAEI Æmilia AEI, given as input, which is randomly selected from the Æmilia architectural specification. From an architectural point of view, the straightforward semantics for CloneAEI is the creation of a copy of srcAEI and the load balancing of incoming requests between the original component and its copy. In order to implement CloneAEI, a proper modification of port types of the source component and its neighbors (i.e. the AEIs which srcAEI directly interacts with) is needed. For example, while cloning a srcAEI, the created clone has to be connected to srcAEI’s neighbors. This implies that involved neighbors’ ports of type UNI (i.e. port type with exactly one connection) have to become OR type (i.e. port type with more than one connection). Nevertheless, the port type modification does not imply any change in the internal neighbors’ behavior. Hence, the CloneAEI action is more complex than similar ones introduced up today in literature (e.g. change of a component multiplicity).

ChangeRate modifies a randomly selected rate of an Æmilia action by multiplying its value by an uniformly distributed FactorOfChange (FOC). ChangeRate intends to represent the option of both enhancing and worsening the performance of a certain action, so that customNSGAII is enabled to find an optimal balance between slower and faster actions. Note that the ChangeRate semantics, in the Æmilia context, is twofold, because it can be charged either to the software contribution of an action (i.e., modifying the complexity of the action) or to the hardware one (e.g., replacing the engine that executes the action with another one); however, this does not modify the semantics of the architecture. The FactorOfChange attribute has been defined as $FOC \in [0.5, 2]$, $FOC \neq 1$, so that the ranges of new actions can go up to double or down to halve the original ones. However, this range can be modified without impact on EASIER.

It is worth to notice that such initial pool of actions does not contain subtractive AEmiliaRefactoringActions as, e.g., the deletion of an AEI. On the one hand, subtractive actions would have enlarged the solution space whereas, on the other hand, they would have introduced a high degree of complexity in both managing the composition of refactoring action sequences during the reproduction phases of the customNSGAII, and in applying the generated sequences. For this reason, we have chosen to focus on non-subtractive refactoring actions, thus leaving subtractive ones as a future research direction.

We have associated ArchDist values to 1.3 and 1 for CloneAEI and ChangeRate, respectively. This is only one possible setting of the distance to which refactoring leads an alternative architecture from the initial one. Other choices can be made, depending on the ADL and the application context.

Finally, in order to generate architectural alternatives that conform to the Æmilia ADL grammar, we have added three constraints onto a refactoring sequence, that are: (i) an AEI cannot be cloned more than once in a sequence, otherwise complex incompatibilities between ports occur; (ii) once cloned an AEI in a sequence, none of its neighbour AEIs can be cloned, due to the same reason as before; (iii) each rate can be changed only once in a sequence, because we want FOC to limit the range of a rate change in each sequence.

ADL-specific refactoring actions are collected into an ADL-dependent Refactoring Actions Library, as shown in

1 The graphical effects of these two actions on an Æmilia architecture are visible by comparing Figs. 8 and 9 in Sec. IV.
An Æmilia architectural specification is basically a textual file (.aem), which conforms to a particular grammar embedding, among topological concepts (i.e., architectural types, instances and connectors), behavioral semantics of architectural elements expressed following an internally defined stochastic process algebra [8]. In order to enable performance analysis of an Æmilia architectural specification, performance measures of interest must be specified (i.e. throughputs and/or utilizations of AEIs/interactions). EASIER currently supports one analysis method, among the ones provided by the Æmilia solver TwoTowers [33], that is Stationary/Transient Reward-Based Measure Calculator. For sake of such type of analysis, performance measures have to be specified into an additional textual file (.rew). Once performance analysis has been completed, indices are available into a textual file (.val).

A further ADL-dependent element to provide for plugging a new ADL in EASIER is represented by PAs detection rules, which had been defined as first-order logics formulae [34]. The current EASIER Æmilia context supports the Pipe&Filter performance antipattern (PaF), which occurs when the slowest filter, in a “pipe and filter” architecture, causes a system interaction to have unacceptable throughput. The original logical formula of PaF in [34] has been implemented as an OCL rule within the Æmilia context as follows: \( \exists i \in I \land \text{rate}(i) \geq \text{Th}_{\text{rateLB}} \land p(i) = 1 \land \text{throughput}(i) < \text{Th}_{\text{throughputUB}} \), where: \( i \) is an element of the set \( I \) of all Architectural Interactions; \( p(i) \) is its probability of execution; \( \text{Th}_{\text{rateLB}} \) and \( \text{Th}_{\text{throughputUB}} \) are threshold values that represent, respectively, a lower bound for \( \text{rate}(i) \) and an upper bound for \( \text{throughput}(i) \), over which the antipattern occurs. The .ocl file implementing the PaF detection rule is evaluated against each generated architectural alternative, and the number of detected PAs is given as input to customNSGAII during the evaluation phases.

2) Process: We recall that EASIER evolutionary context works on solutions containing a reference to the corresponding architectural alternative resulting from the application of the genome refactoring sequence and conforming to an ADL metamodel. Hence, in case of Æmilia, textual specifications need to undergo a technical in-place text-to-model transformation from the Æmilia grammar to the Æmilia metamodel. A model-based infrastructure was proposed in [35] to this goal, providing the Æmilia metamodel and the text-to-model transformation from textual specifications to models (.mmaemilia) conforming to such metamodel. These model-based features are provided within a single environment, in the context of a pre-existing open-source tool named TwoEagles [36].

As required in the evolutionary context, we have implemented a bridge, namely TwoEaglesBridge, between customNSGAII and TwoEagles. TwoEaglesBridge allows to obtain an Æmilia model annotated with performance indices. In particular, for each candidate solution generated during customNSGAII execution, the bridge executes the following automated steps: (i) It applies the refactoring to the initial architecture, thus resulting in a Refactored Architectural Alternative (.mmaemilia); (ii) It applies the model-to-text transformation of the obtained Æmilia model (.mmaemilia), which produces a corresponding textual file (.aem) processed by TwoTowers; (iii) It executes the performance analysis, that takes as input .aem and .rew files, and computes performance indices (.val file); (iv) It fills the obtained performance indices (namely Measures To Indices) into the model reported by the evolutionary context to enable the computation of \( \text{PerfQ} \) and \#PAs. (v) It updates the refactoring genome.

IV. Validation

In order to validate EASIER, we have applied a benchmark of different configurations for customNSGAII to the Æmilia specification of a Fire Tracking System (FTS).

FTS represents a software system that monitors a building to timely locate an indoor fire and to support firefighters in case rescue actions are needed. A Wireless Sensors Network (WSN) monitors the environment with a certain sampling rate (namely workload). Sample data, through a channel (CHN), are stored in a database (DB) and analyzed by a Fire Tracking Application central unit (FTA), which triggers an alarm in case of a risky situation. Secure communications between WSN and the firefighter’s desktop application (called DSK) are guaranteed, hence data collected from the building need to be decrypted before being processed by FTA. Decryption is performed by a security component (SCR), and data are then forwarded back to the FTA, through a LAN that connects FTA to DB, DSK and SRC.

A. Initial Architecture

While it is obvious that an initial architecture is created in an architectural design phase, not as much obvious it is for an existing system. For sake of this paper, we target the former case, whereas reverse engineering techniques shall be adopted to apply our approach to the latter case.

Fig. 3 shows the FTS as an Æmilia flow graph. Rectangles represent Æmilia Architectural Element Instances (AEIs), whereas squares represent Æmilia interactions, namely “ports” between the system and the environment or between two AEIs. Moreover, arrows represent communication links between different interactions, while non-connected interactions represent either external ports (e.g., the workload one) or components’ internal actions (e.g.,\( \text{ftu\_rate} \) and \( \text{packet\_rate} \)). The dashed arrows represent the internal flow of components. Since the FTA behavior is rather complex, for sake of simplicity we have labeled its internal flow sequence with numbers. Note that the rate of each output and
each internal processing interaction is shown in the figure. Unlabeled interactions are passive ones, hence they do not have a rate. Practically, this means that only the labeled ones can be modified by the ChangeRate refactoring action.

In order to evaluate PerfQ and #PAs objectives, a performance analysis of each generated solution is needed, and this introduces a time overhead that increases as the complexity of the architectural specification increases. Hence, we have defined an upper bound on the execution time of each run, called plausibility threshold, representing the maximum time beyond which the execution time of a run is considered non-plausible and is stopped. For FTS case study, we have set a plausibility threshold of 12 hours, as driven by trial runs. Only the executions with #evals = 640 have violated this threshold, and therefore they do not appear in our results.

We have also introduced a further configuration parameter, that is the maximum number of CloneAEI occurrences into a genome sequence, namely maxCloning. It gives more flexibility to our approach, because the cloning operation obviously affects the time required for performance analysis. In particular, we have observed that the time spent on performance analysis over-linearly increases as the number of AEI clones increases. Therefore, again based on trial runs, in our scenario we have found suited to set maxCloning = 3. Finally, threshold values for PAs detection have to be tuned. In particular, we have set $T_{rateLB} = 3.486$ and $T_{throughputUB} = 0.162$, corresponding to average values of rates and interactions throughputs, respectively.

2) Results: The results of our experimental evaluation are shown in Table I where runs are ordered in #evals ascending order. In details, Sol ID (1st column) identifies the configuration as #epo-pop:id, where id is assigned by EASIER to the solution. All Pareto-optimal solutions of each configuration are reported. In particular, for each configuration we report the values of the fitness function objectives (2nd, 3rd, and 4th columns). We also provide details of the four refactoring actions associated to each solution genome in the Pareto front (columns 5th to 16th). In details, Sol ID (1st column) identifies the configuration as #epo-pop:id, where id is assigned by EASIER to the solution. All Pareto-optimal solutions of each configuration are reported. In particular, for each configuration we report the values of the fitness function objectives (2nd, 3rd, and 4th columns). We also provide details of the four refactoring actions associated to each solution genome in the Pareto front (columns 5th to 16th). In particular, each action in the sequence of a Pareto solution is identified by its type (i.e. CloneAEI or ChangeRate), its target element (i.e. the cloned AEI or the modified rate) and the applied FOC (in case of ChangeRate).

3) Analysis: In what follows, we analyze the experimental results w.r.t. the following aspects: (i) execution times, (ii) quality (i.e. fitness function values) of the Pareto solutions, (iii) suggested sequences of refactoring.

Execution time: Our data have shown us that, on average, a single run took 5297s while a single solution required 27.53s of execution time. In particular, we have
Table I: Benchmark results.

<table>
<thead>
<tr>
<th>Sol ID</th>
<th>Fitness function values</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Action type</td>
<td>Target element</td>
<td>FOC</td>
<td>Action type</td>
</tr>
<tr>
<td>5-4:4</td>
<td>0.008 4.6 1</td>
<td>ChangeRate</td>
<td>channel rate</td>
<td>1.930</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>5-8:1</td>
<td>0.102 4.3 2</td>
<td>CloneAEI</td>
<td>LAN</td>
<td>-</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>5-8:59</td>
<td>0.017 4.6 1</td>
<td>CloneAEI</td>
<td>DB</td>
<td>-</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>10-4:10</td>
<td>0.146 4.3 5</td>
<td>ChangeRate</td>
<td>lan_rate</td>
<td>1.625</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>10-4:23</td>
<td>0.033 4.6 2</td>
<td>CloneAEI</td>
<td>LAN</td>
<td>-</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>10-4:35</td>
<td>0.162 4.6 3</td>
<td>ChangeRate</td>
<td>dec_rate</td>
<td>1.928</td>
<td>CloneAEI</td>
</tr>
<tr>
<td>10-4:28</td>
<td>0.197 4.3 2</td>
<td>CloneAEI</td>
<td>CHN</td>
<td>-</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>5-16:3</td>
<td>0.206 4.0 5</td>
<td>ChangeRate</td>
<td>lan_rate</td>
<td>1.691</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>5-16:13</td>
<td>0.134 4.0 1</td>
<td>ChangeRate</td>
<td>lan_rate</td>
<td>0.897</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>10-8:12</td>
<td>0.076 4.0 5</td>
<td>ChangeRate</td>
<td>link_rate</td>
<td>1.725</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>10-8:15</td>
<td>0.004 4.0 2</td>
<td>ChangeRate</td>
<td>lan_rate</td>
<td>1.956</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>10-8:12</td>
<td>0.164 4.0 3</td>
<td>ChangeRate</td>
<td>lan_rate</td>
<td>1.600</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>10-8:63</td>
<td>0.155 4.3 2</td>
<td>ChangeRate</td>
<td>lan_rate</td>
<td>0.541</td>
<td>CloneAEI</td>
</tr>
<tr>
<td>20-4:40</td>
<td>0.186 4.0 1</td>
<td>ChangeRate</td>
<td>wsn_rate</td>
<td>0.870</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>5-32:29</td>
<td>0.193 4.0 2</td>
<td>CloneAEI</td>
<td>SCR</td>
<td>0.566</td>
<td>CloneAEI</td>
</tr>
<tr>
<td>5-32:5</td>
<td>0.171 4.6 2</td>
<td>ChangeRate</td>
<td>packet_rate</td>
<td>0.813</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>5-32:48</td>
<td>0.167 4.3 1</td>
<td>ChangeRate</td>
<td>CHN</td>
<td>-</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>10-16:194</td>
<td>0.274 4.6 1</td>
<td>ChangeRate</td>
<td>dec_rate</td>
<td>0.519</td>
<td>CloneAEI</td>
</tr>
<tr>
<td>10-16:241</td>
<td>0.287 4.6 4</td>
<td>ChangeRate</td>
<td>dec_rate</td>
<td>1.925</td>
<td>CloneAEI</td>
</tr>
<tr>
<td>10-16:105</td>
<td>0.192 4.6 2</td>
<td>ChangeRate</td>
<td>wsn_rate</td>
<td>0.892</td>
<td>CloneAEI</td>
</tr>
<tr>
<td>10-16:265</td>
<td>0.171 4.6 1</td>
<td>ChangeRate</td>
<td>wsn_rate</td>
<td>1.986</td>
<td>CloneAEI</td>
</tr>
<tr>
<td>20-8:147</td>
<td>0.186 4.0 3</td>
<td>ChangeRate</td>
<td>lan_rate</td>
<td>1.501</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>20-8:245</td>
<td>0.293 4.6 1</td>
<td>CloneAEI</td>
<td>SCR</td>
<td>-</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>10-32:505</td>
<td>0.293 4.6 1</td>
<td>CloneAEI</td>
<td>SCR</td>
<td>-</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>20-16:504</td>
<td>0.281 4.6 4</td>
<td>CloneAEI</td>
<td>FTA</td>
<td>-</td>
<td>ChangeRate</td>
</tr>
<tr>
<td>20-16:349</td>
<td>0.297 4.6 1</td>
<td>CloneAEI</td>
<td>SCR</td>
<td>-</td>
<td>ChangeRate</td>
</tr>
</tbody>
</table>
observed that TwoTowers resulted (as expected) to be the most time-consuming component of EASIER, as it has taken around 69% of the 5826s total running time.

Quality of Solutions: First, we observe that the average size of the Pareto frontier is 2.36 solutions per configuration. Moreover, w.r.t. the three objectives, we noticed that in the resulting set of Pareto solutions: i) the maximum (average, resp.) value of $PerfQ$ is 0.29 (0.17, resp.); ii) the minimum (average, resp.) value of $ArchDist$ is 4.00 (4.36, resp.), which means slightly more than one CloneAEI per element of the Pareto set, on average; iii) the minimum (average, resp.) value of $#PAs$ is 1.00 (2.23, resp.), against the value of 3 in the original FTS, which means a decreasing of about 26%, on average.

In addition, our experiments show that the solutions found by EASIER, in terms of suggested combinations of refactoring actions, are rather different by each other, i.e. EASIER provides diversity in the Pareto frontier. In particular, $\frac{24}{26} \approx 0.92 = 92\%$ solutions were pairwise different by at least one refactoring action in the sequence.

We like to remark that, as shown in Fig. 4, $PerfQ$ follows a logarithmic trend vs. the number of iterations $#evals$, while $ArchDist$ and $#PAs$ objectives do not exhibit such a distinguished relationship. This suggests that EASIER is able to go towards global optima only for $PerfQ$ while extending $#evals$, as we will discuss further in this section.

![Figure 4: Trend of objectives vs number of evaluations.](image)

A more detailed comparison of all Pareto solutions is drawn in Fig. 5, where: $\frac{1}{ArchDist}$ is on the x-axis, $PerfQ$ is on the y-axis, $\frac{1}{#PAs}$ determines the bubble size and bubbles having the same color belong to the same frontier. Therefore, best solutions are expected to be large bubbles residing in the top-right corner of the figure. In addition, boxes with bolded labels define a “super” Pareto frontier, containing non-dominated solutions overall the whole benchmark, which are thus likely to reach a global optima.

At the top-left corner, solutions with the highest $PerfQ$ occur, which have been obtained with the highest $#evals$, i.e. 160 and 320, and this is coherent with the trend of $PerfQ$ shown in Fig. 4. These solutions also show highest values of $ArchDist$, thus inducing the idea that, in order to sensibly improve the architecture performance it is necessary to largely deviate from the initial architecture. As a side observation, most of such solutions also exhibit the lowest values of $#PAs$.

On the right side of the figure, we observe solutions having the lowest values of $ArchDist$, which can be of particular interest for software architects that search for low distance solutions that minimally impact on refactoring effort and complexity. Although most of these solutions have been obtained with a low number of evaluations (i.e., $#evals = 80$), and they show a $PerfQ$ which is at least 40% lower than the highest values, a considerable amount among them lead a 20% performance improvement. This can be of great interest in cases where such improvement can be satisfactory for software architects, because these solutions nicely combine this characteristic with a low distance from the initial architecture.

Suggested actions: In Table I we can observe that, in general, the more frequent refactoring actions are actually the more effective ones in terms of performance. In particular, the most effective action is definitely the dec_rate increment (often doubled up), followed by the cloning of the SCR component it belongs to. This is in line with the fact that SCR actually represents a potential bottleneck in the initial FTS, because its rate is the lowest one. Moreover, we observe that each time the SCR cloning is suggested, the FTA cloning is suggested as well. Although this is difficult to come by intuition, as for SCR, it seems a reasonable implication. In fact, FTA is the unique AEI communicating with SCR, and the rate of the LAN in the middle is not sufficiently high to manage their communication rates at the SCR-side (i.e. lan_sec_rate increment). Besides, we observe that CHN cloning has been mostly suggested by running EASIER with a small $#evals$ (e.g. 40) and the obtained $PerfQ$ very likely depend on the other actions in the sequences with CHN.

For sake of illustration, in Fig. 6, we draw the flow graph of a Pareto solutions from Table I, i.e. the grey row of the table.

C. Threats to validity

In this section we identify the major limitations of EASIER, which might induce potential threats to its validity.

Experimental setting: our experimental results show that, in the Æmilia context, EASIER spends most of the time on performance analyses, i.e. while waiting for TwoEagles to compute $PerfQ$ and $#PAs$. To mitigate this threat, we have introduced a plausibility threshold in the experimental setup (see Section IV-B1).

Generalization of results: We have widely tested the approach on a single Æmilia architectural specification,
hence we do not know its effectiveness on other case study, even though the current results are promising. However, the FTS architecture used here has been selected, for size and quality, out of around 30 graduate student projects.

Refactoring actions: In this paper we have assumed predefined, fixed, $ArchDist$ for refactoring actions. This is an aspect that needs more investigation, by providing some guidelines to set these values in different ADL contexts and, possibly, in different application domains. Moreover, in the current Æmilia Refactoring Actions Library, only additive actions are considered. This may represent a limitation in the refactoring possibilities. However, designing and implementing delete actions is a complex task, due to the fact that they heavily impact on the feasibility of a refactoring in terms of pre and post conditions.

Pareto solutions: A multiobjective optimization process has two main goals: 1) convergence to the Pareto-optimal set, and 2) diversity in the considered intermediate solutions. Our results suggest that diversity is maintained within the Æmilia context and that the longer customNSGAII runs (i.e. $#evals$ increases), the more the performance quality of Pareto solutions increases. However, a deeper investigation of these two aspects will be needed to assess the robustness of our approach.

PA detection: Thresholds calibration is a crucial task for PAs detection. Adopting a simplistic strategy (i.e. based on average values), like we did in this paper, however, does not jeopardize EASIER validity, since adding complexity to thresholds calibration would just result in more inclusive/exclusive detection policies. However, EASIER current implementation does not calculate thresholds for architectural alternatives before counting the PAs occurring on the latter one, as it always applies the original thresholds for PA detection. This might affect the precision of the detection procedure, and it is very likely the motivation for the fluctuation in the values of $#PAs$ across Pareto solutions. Therefore, threshold (re-)calibration is certainly an issue to be investigated in future.

V. Conclusion

In this paper we have presented EASIER, an evolutionary approach for architecture refactoring based on performance aspects. The first experimental results of our approach are promising in terms of its applicability in practice. Beyond the directions that could mitigate the threats to validity introduced in Section IV-C, we also intend to pursue the following objectives in the future: (i) Experimentation while scaling over the architecture size and across different values of its main parameters, with a particular emphasis on the genome length; (ii) Implementation of more performance antipatterns detection rules and an automated mechanism that supports such implementation process; (iii) Validation
of the approach over different ADL contexts; (iv) Extension of the fitness function to metrics related, for example, to budget aspects (e.g. refactoring cost), as well as to other non-functional properties (e.g. reliability).

REFERENCES


