EASIER: Evolutionary Approach for multi-objective Software archItecturE Refactoring

Davide Arcelli, Vittorio Cortellessa, Mattia D'Emidio and
Daniele Di Pompeo

January 12, 2019

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 aimed at leading to optimal architectural alternatives from an initial one. Moreover, we introduce in this context the knowledge dwelling in performance antipatterns [6] for supporting the search process. Performance antipatterns are, in fact, well-known bad practices that induce performance degradation. As an initial study in this direction, we intend to observe whether their introduction could improve the search process.

Fig. 1 illustrates the EASIER high-level architecture. The EASIER core is represented by a custom NSGA-II algorithm [7], namely `customNSGAII`, that takes as input an initial software architecture and searches the architectural space by (re-)combining 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 a performance quality indicator, 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.
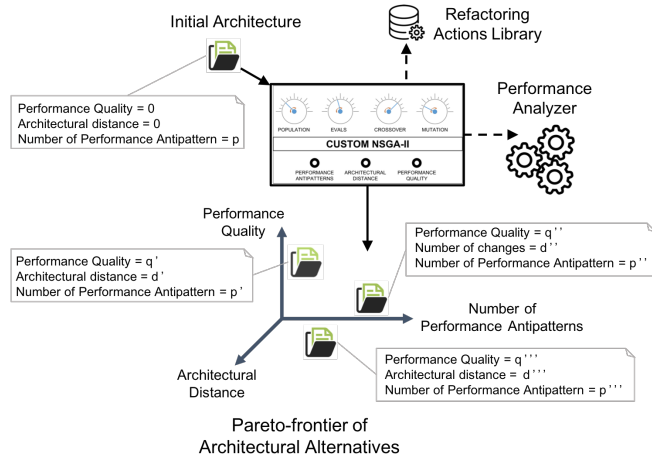

Figure 1: The EASIER high-level architecture.

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 [3], [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

architectural details and resource contention.

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 [30]. We do not provide details-bec 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-B1.

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**
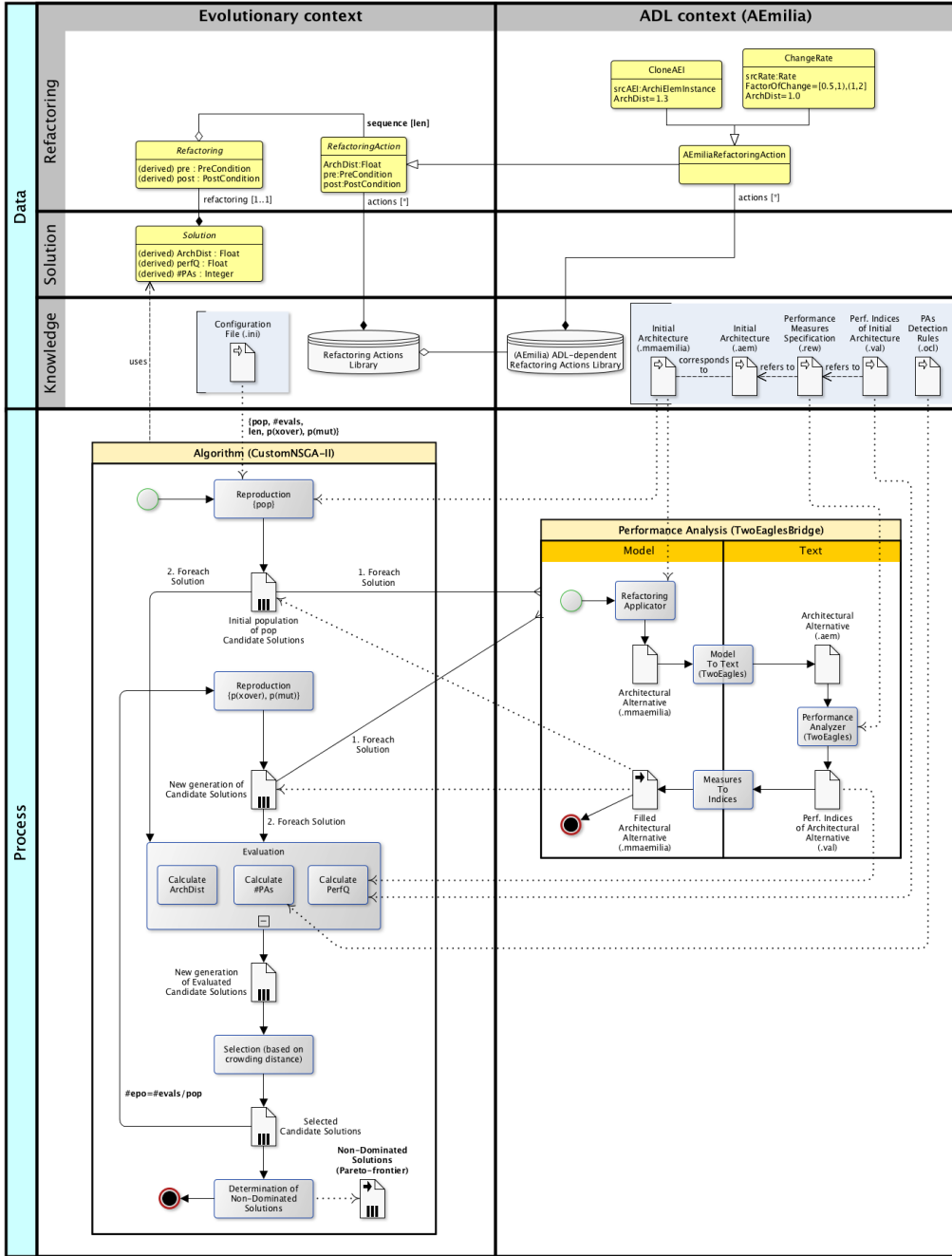
Figure 2: The EASIER low-level architecture.

population of *candidate solutions* (i.e. refactorings by len length) with a *pop* cardinality. This phase is a customized step where, every time `customNSGAII` 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* [30].

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 $PerfQ(A) = \frac{1}{c}(\sum_{j=1}^{c} p \cdot (\frac{F_j - I_j}{F_j + I_j}))$, 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 $PerfQ(A)$ 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 $PerfQ(A)$ will be a positive real only when the majority of the terms contribute with positive values. For this reason, we consider $PerfQ$ 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 [26], we define a *correction factor* Δ_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:

$$\Delta_j = \begin{cases} -2\frac{F_j - I_j}{F_j + I_j} & \text{if } F_j > 0.8 \wedge I_j > 0.8 \\ 0.8 - F_j & \text{if } F_j > 0.8 \wedge I_j \le 0.8 \\ I_j - 0.8 & \text{if } F_j \le 0.8 \wedge I_j > 0.8 \\ 0 & \text{otherwise} \qquad \square \end{cases}$$

$\underline{ArchDist}$ (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-

ing genome. Note that, in the current version of EASIER, $ArchDist(a_i)$ is assumed to be predefined for each `RefactoringAction`. The setting of these values is left to software architects, because they might depend on the characteristics of the specific ADL and/or the application domain. $\qquad \square$

$\underline{\#PAs}$ (to minimize). It counts the number of performance antipatterns (PAs) occurrences within an alternative architecture. In its current version, EASIER supports OCL [31] for antipatterns detection rules specification and verification, which represents yet another entry point. In fact, these rules depend on the ADL expressiveness, thus they must be provided when plugging a new ADL context into EASIER.

To the best of our knowledge, EASIER is the first approach that considers the number of PAs as an objective of an evolutionary algorithm's fitness function. Our intent is to start investigating whether PAs in EASIER can play an analogous role of, respectively, architectural tactics in [24] and architectural patterns in [26], that are guiding the search process with additional architectural knowledge. In our case they are negative practices to be avoided, whereas in the other two cases were positive practices to be adopted. $\square$

After the (custom) evaluation, non-dominated solutions (i.e. solutions that are better than all others w.r.t. at least one objective) are **ranked** according to the notion of crowding distance [7], and the best ones among them are then **selected** and used as *reproductive basis* for the next iterations. In particular, in each subsequent iteration, new sets of candidate solutions are generated by randomly applying (custom) *crossover* and *mutation* operators to the reproductive basis of the previous step, with probabilities $p(xover)$ and $p(mut)$.

Concerning crossover, in EASIER a **(custom) operator** is applied onto two parent `Refactorings` r_1 and r_2 selected by tournament [32]. The crossover point x is chosen (uniformly at random) to be an integer value within the range $[1, len-1]$. Then, two children are generated by single-point crossover-based strategy as follows: the first x actions of r_1 are combined with the second $len - x$ actions of r_2 and viceversa, as long as the x–th action of r_1 is *compatible* with the $(x + 1)$–th action of r_2. On the contrary, the child is discarded and one of the parents replaces it.

Concerning mutation, in EASIER we defined a **(custom) operator** that randomly choses a `RefactoringAction` of the considered `Refactoring` and replaces it with another *compatible*, randomly generated, `RefactoringAction`.

After the application of crossover and mutation operators, the obtained candidate solutions are in turn **evaluated** and **selected** for the next iteration, as for the initial population. The process proceeds for a number epo of iterations that is given by $evals/pop$. After epo iterations, the available solutions are once more compared each other and the set of

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 EAS-IER. 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 `AEmiliaRefactoringAction` 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 implies 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 [1].

It is worth to notice that such initial pool of actions does not contain subtractive `AEmiliaRefactoringAction`s 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 Figg. 3 and 6 in Sec. IV

Fig. 2.

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 [6]. The original logical formula of PaF in [34] has been implemented as an OCL rule within the Æmilia context as follows:$\exists i \in \mathrm{I}, |\ rate(i) \geq Th_{rateLB} \wedge p(i) = 1 \wedge throughput(i) < Th_{throughputUB}$, where: i is an element of the set I of all Architectural Interactions; $p(i)$ is its probability of execution; Th_{rateLB} and $Th_{throughputUB}$ are threshold values that represent, respectively, a lower bound for $rate(i)$ and an upper bound for $throughput(i)$, 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 $PerfQ$ 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., *fta_rate* and *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.
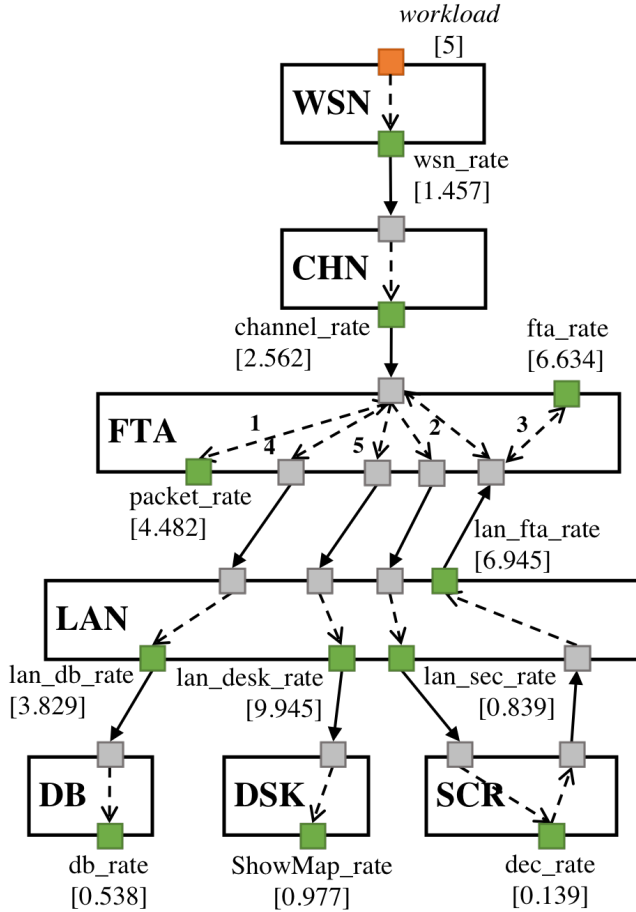

Figure 3: Æmilia flow graph of the Fire Tracking System.

B. Experimentation

In this section, we first describe our experimental setup, and then the obtained results are reported and analyzed.

1) Setup: Among the customizable parameters of `customNSGAII`, on the one hand, we focus on $\#epo$, pop, and $\#evals$, since their tuning affects the distance of the obtained solutions from the initial architecture. On the other hand, we do not vary $p(xover)$ and $p(mut)$, as they are essentially related to the behavior of `customNSGAII` only. In particular, we set $p(xover) = 0.8$ and $p(mut) = 0.2$ based on their wide adoption in literature [37], as well as a number of our trial runs (in the order of one hundred). For sake of this paper, we have chosen $\#epo \in \{5, 10, 20\}$ and $\#pop \in \{4, 8, 16, 32\}$. As a result, we have obtained 12 different configurations, with $\#evals \in \{20, 40, 80, 160, 320, 640\}$.

The len parameter represents the genome length. This is an important degree of freedom of EASIER, in that

although long sequences of refactoring could lead to better solutions in terms of performance, they could also lead to architectural alternatives very distant from the initial one. We have set $len = 4$ for sake of this paper experimentation, as it has seemed (from trial runs) a good compromise between execution time and quality of solutions. However, a larger experimentation to study the effect of this parameter would be very interesting, and it is part of our future work.

In order to evaluate $PerfQ$ and $\#PAs$ objectives, a performance analysis of each generated solutions 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 $Th_{rateLB} = 3.486$ and $Th_{throughputUB} = 0.162$, 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 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 1: Benchmark results.

| Sol ID | Fitness function values | | | Actions | | | | | | | | | | | |
| | PerfQ | archDist | #PAs | 1 | | | 2 | | | 3 | | | 4 | | |
				Action type	Target element	FOC	Action type	Target element	FOC	Action type	Target element	FOC	Action type	Target element	FOC
5-4:4	0.008	4.6	1	ChangeRate	channel_rate	1.930	ChangeRate	packet_rate	0.580	CloneAEI	CHN	-	CloneAEI	SCR	-
5-8:1	0.102	4.3	2	CloneAEI	LAN	-	ChangeRate	lan_sec_rate	1.412	ChangeRate	dec_rate	1.866	ChangeRate	db_rate	1.817
5-8:59	-0.017	4.6	1	CloneAEI	DB	-	ChangeRate	lan_desk_rate	1.652	CloneAEI	CHN	-	ChangeRate	packet_rate	1.639
10-4:10	0.146	4.3	5	ChangeRate	lan_desk_rate	1.625	ChangeRate	lan_db_rate	1.607	CloneAEI	CHN	-	ChangeRate	dec_rate	1.792
10-4:23	0.033	4.6	2	CloneAEI	LAN	-	ChangeRate	lan_db_rate	1.001	CloneAEI	CHN	-	ChangeRate	dec_rate	1.574
10-4:65	0.162	4.6	3	ChangeRate	dec_rate	1.928	CloneAEI	CHN	-	CloneAEI	DB	-	ChangeRate	lan_fta_rate	1.860
10-4:28	0.197	4.3	2	CloneAEI	CHN	-	ChangeRate	lan_sec_rate	1.460	ChangeRate	lan_desk_rate	1.042	ChangeRate	dec_rate	1.988
5-16:83	0.206	4.0	5	ChangeRate	lan_fta_rate	1.691	ChangeRate	lan_db_rate	0.703	ChangeRate	lan_sec_rate	1.759	ChangeRate	dec_rate	1.996
5-16:113	0.134	4.0	1	ChangeRate	lan_fta_rate	0.897	ChangeRate	lan_db_rate	1.250	ChangeRate	db_rate	0.684	ChangeRate	dec_rate	1.654
10-8:122	0.076	4.0	5	ChangeRate	lan_sec_rate	1.735	ChangeRate	db_rate	0.619	ChangeRate	fta_rate	0.548	ChangeRate	dec_rate	1.254
10-8:115	0.004	4.0	2	ChangeRate	lan_sec_rate	1.573	ChangeRate	db_rate	1.956	ChangeRate	fta_rate	1.032	ChangeRate	lan_db_rate	0.868
10-8:125	0.164	4.0	3	ChangeRate	ShowMap_rate	1.660	ChangeRate	lan_fta_rate	1.221	ChangeRate	dec_rate	1.870	ChangeRate	fta_rate	1.318
10-8:63	0.155	4.3	2	ChangeRate	channel_rate	0.541	CloneAEI	DSK	-	ChangeRate	dec_rate	1.989	ChangeRate	fta_rate	1.153
20-4:40	0.186	4.0	1	ChangeRate	wsn_rate	0.870	ChangeRate	dec_rate	1.910	ChangeRate	channel_rate	1.342	ChangeRate	lan_db_rate	1.284
5-32:29	0.193	4.0	2	CloneAEI	SCR	-	ChangeRate	packet_rate	0.566	CloneAEI	FTA	-	ChangeRate	ShowMap_rate	1.377
5-32:5	0.171	4.6	2	ChangeRate	packet_rate	0.813	ChangeRate	db_rate	1.052	ChangeRate	lan_fta_rate	0.858	CloneAEI	FTA	-
5-32:48	0.167	4.3	1	ChangeRate	CHN	-	ChangeRate	lan_db_rate	0.545	ChangeRate	fta_rate	0.773	ChangeRate	dec_rate	1.932
10-16:194	0.274	4.6	1	ChangeRate	dec_rate	0.519	CloneAEI	FTA	-	ChangeRate	lan_desk_rate	1.977	CloneAEI	SCR	-
10-16:241	0.287	4.6	4	ChangeRate	dec_rate	1.925	CloneAEI	FTA	-	CloneAEI	SCR	-	ChangeRate	lan_db_rate	1.355
10-16:208	0.184	4.6	2	ChangeRate	wsn_rate	0.642	CloneAEI	FTA	-	CloneAEI	SCR	-	ChangeRate	lan_sec_rate	1.777
10-16:263	0.171	4.6	1	ChangeRate	wsn_rate	1.986	CloneAEI	FTA	-	CloneAEI	SCR	-	ChangeRate	lan_sec_rate	1.933
20-8:147	0.186	4.0	3	ChangeRate	lan_db_rate	1.501	ChangeRate	dec_rate	1.929	ChangeRate	lan_fta_rate	0.894	ChangeRate	fta_rate	0.963
20-8:245	0.293	4.6	1	CloneAEI	SCR	-	ChangeRate	FTA	-	ChangeRate	dec_rate	1.876	ChangeRate	lan_sec_rate	1.223
10-32:505	0.293	4.6	1	CloneAEI	SCR	-	ChangeRate	dec_rate	1.922	CloneAEI	FTA	-	ChangeRate	fta_rate	0.625
20-16:504	0.281	4.6	4	CloneAEI	FTA	-	ChangeRate	fta_rate	1.207	CloneAEI	SCR	-	ChangeRate	dec_rat	1.904
20-16:349	0.279	4.6	1	CloneAEI	SCR	-	CloneAEI	FTA	-	ChangeRate	ShowMap_rate	0.089	ChangeRate	dec_rate	1.814

observed that TwoTowers resulted (as expected) to be the most time-consuming component of EASIER, as it has taken around 69% of the $58262s$ 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.
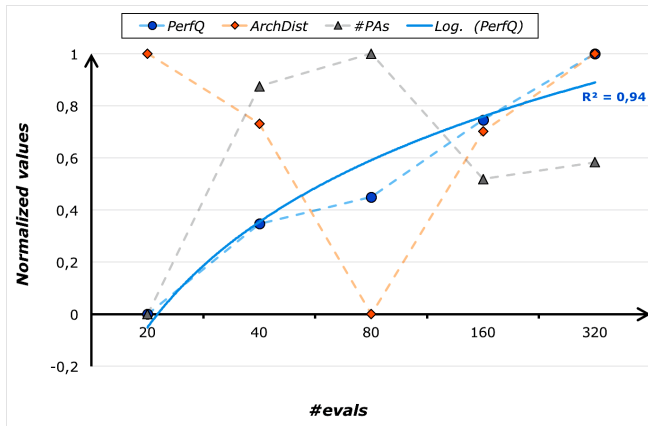

Figure 4: Trend of objectives vs number of evaluations.

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,
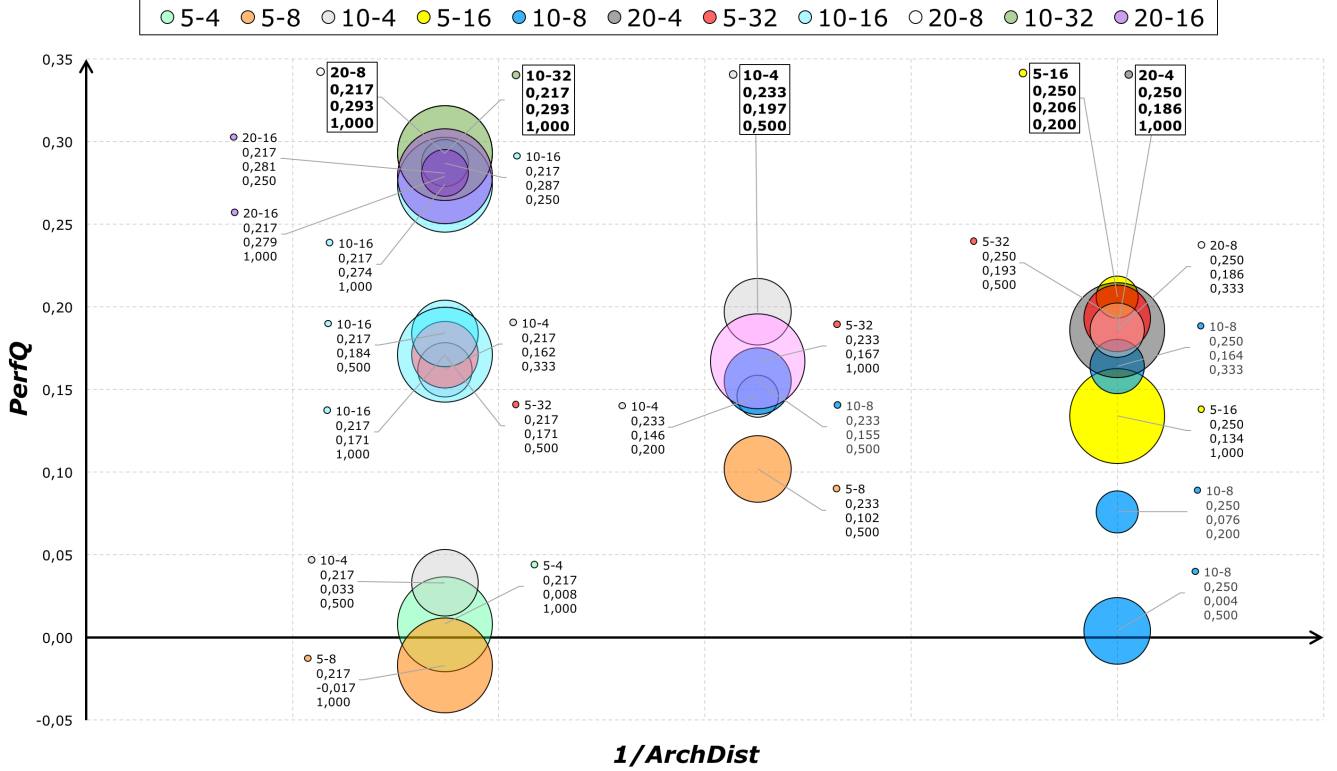
Figure 5: Comparison of Pareto frontiers across different runs.

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 [7]. 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
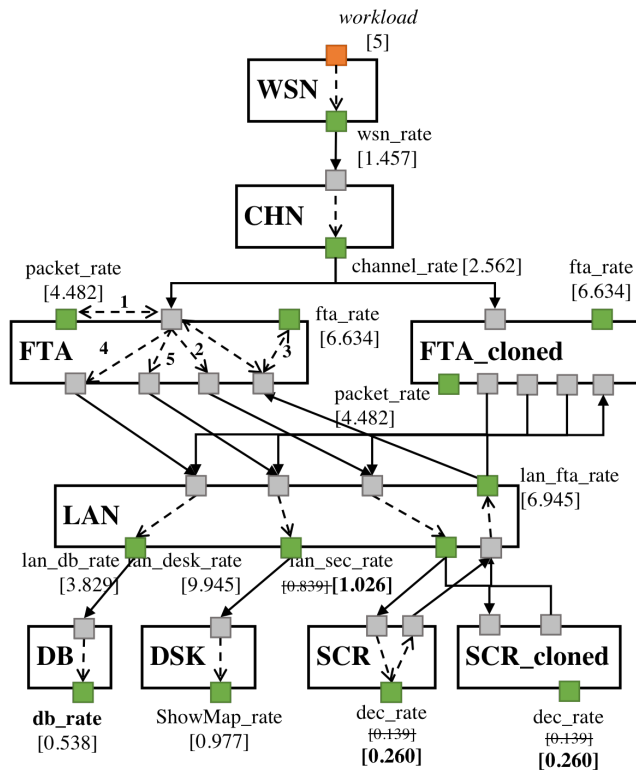
Figure 6: Flow graph of Sol ID : `20-8:245` from Table I.

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

[1] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software Architecture Optimization Methods: A Systematic Literature Review," *IEEE Trans. on Softw. Eng.*, vol. 39, no. 5, pp. 658–683, 2013.

[2] V. Cortellessa, A. D. Marco, and P. Inverardi, *Model-Based Software Performance Analysis*. Springer, 2011.

[3] A. Martens, H. Koziolek, S. Becker, and R. Reussner, "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms," in *ICPE*, 2010, pp. 105–116.

[4] A. Aleti, S. Björnander, L. Grunske, and I. Meedeniya, "Archeopterix: An extendable tool for architecture optimization of AADL models," in *MOMPES*, 2009, pp. 61–71.

[5] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, 2003.

[6] C. U. Smith and L. G. Williams, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," *Computer Measurement Group Conference*, 2003.

[7] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, pp. 1–16, Apr. 2002.

[8] M. Bernardo, L. Donatiello, and P. Ciancarini, "Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language," in *Performance Evaluation of Complex Systems Techniques and Tools*, 2002, pp. 236–260.

[9] M. De Sanctis, C. Trubiani, V. Cortellessa, A. Di Marco, and M. Flamminj, "A model-driven approach to catch performance antipatterns in ADL specifications," *Inf. Softw. Technol.*, vol. 83, pp. 35–54, 2017.

[10] W. G. Griswold and W. F. Opdyke, "The birth of refactoring: A retrospective on the nature of high-impact software engineering research," *IEEE Software*, vol. 32, no. 6, pp. 30–38, 2015.

[11] D. Arcelli, V. Cortellessa, and D. D. Pompeo, "Performance-driven software model refactoring," *Inf. Softw. Technol.*, vol. 95, pp. 366 – 397, 2018.

[12] A. Ghannem, G. El-Boussaidi, and M. Kessentini, "Model refactoring using interactive genetic algorithm," in *SSBSE*, 2013, pp. 96–110.

[13] M. Misbhauddin and M. Alshayeb, "UML model refactoring - a systematic literature review," *Empir. Softw. Eng.*, vol. 20, no. 1, pp. 206–251, 2015.

[14] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *JIST*, vol. 83, pp. 14 – 34, 2017.

[15] M. Mohan, D. Greer, and P. McMullan, "Technical debt reduction using search based automated refactoring," *J. Syst. Softw.*, vol. 120, pp. 183–194, 2016.

[16] R. Mahouachi, M. Kessentini, and M. Ó Cinnéide, "Search-Based Refactoring Detection Using Software Metrics Variation," in *SSBSE*, 2013, pp. 126–140.

[17] U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb, "Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm," pp. 1–29, 2015.

[18] A. Ouni, M. Kessentini, H. Sahraoui, M. Ó. Cinnéide, K. Deb, and K. Inoue, "A Multi-Objective Refactoring Approach to Introduce Design Patterns and Fix Anti-Patterns," in *SSBSE*, 2015, pp. 1–16.

[19] R. Li, R. Etemaadi, M. T. M. Emmerich, and M. R. V. Chaudron, "An evolutionary multiobjective optimization approach to component-based software architecture design," in *IEEE CEC*, 2011, pp. 432–439.

[20] I. Meedeniya, B. Buhnova, A. Aleti, and L. Grunske, "Architecture-driven reliability and energy optimization for complex embedded systems," in *QoSA*, 2010, pp. 52–67.

[21] A. Martens, D. Ardagna, H. Koziolek, R. Mirandola, and R. Reussner, "A hybrid approach for multi-attribute qos optimisation in component based software systems," in *QoSA*.

[22] F. Rosenberg, M. B. Müller, P. Leitner, A. Michlmayr, A. Bouguettaya, and S. Dustdar, "Metaheuristic optimization of large-scale qos-aware service compositions," in *IEEE SCC*, 2010, pp. 97–104.

[23] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola, "Qos-driven runtime adaptation of service oriented architectures," in *ESEC/FSE*, 2009, pp. 131–140.

[24] A. Koziolek, H. Koziolek, and R. Reussner, "PerOpteryx: automated application of tactics in multi-objective software architecture optimization," in *QoSA*, 2011, pp. 33–42.

[25] S. Becker, H. Koziolek, and R. H. Reussner, "The Palladio component model for model-driven performance prediction." *JSS*, 2009.

[26] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malek, and J. P. Sousa, "A framework for utility-based service oriented design in SASSY," in *WOSP/SIPEW*, 2010, pp. 27–36.

[27] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language*, ser. SEI series in software engineering. Addison-Wesley, 2012.

[28] A. J. Nebro, J. J. Durillo, and M. Vergne, "Redesigning the jmetal multi-objective optimization framework," in *GECCO Companion*, 2015, pp. 1093–1100.

[29] A. S. Sayyad, T. Menzies, and H. Ammar, "On the value of user preferences in search-based software engineering: A case study in software product lines," in *ICSE*, 2013, pp. 492–501.

[30] M. Ó Cinnéide and P. Nixon, "Composite refactorings for Java programs," in *European Conference Object-Oriented Programming*, 2000.

[31] J. B. Warmer and A. G. Kleppe, *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.

[32] A. Eiben and S. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms," *Swarm Evol Comput*, vol. 1, no. 1, pp. 19 – 31, 2011.

[33] M. Bernardo, "Twotowers: User manual." [Online]. Available: http://www.sti.uniurb.it/bernardo/twotowers/manual.pdf

[34] V. Cortellessa, A. Di Marco, and C. Trubiani, "An approach for modeling and detecting software performance antipatterns based on first-order logics," *SoSyM*, vol. 13, no. 1, pp. 391–432, 2014.

[35] V. Cortellessa, M. D. Sanctis, A. D. Marco, and C. Trubiani, "Enabling performance antipatterns to arise from an adl-based software architecture," in *WICSA/ECSA*, 2012, pp. 310–314.

[36] M. Bernardo, V. Cortellessa, and M. Flamminj, "TwoEagles: A Model Transformation Tool from Architectural Descriptions to Queueing Networks," in *EPEW*, 2011, pp. 265–279.

[37] A. Arcuri and G. Fraser, "Parameter tuning or default values? An empirical investigation in search-based software engineering," *Empir. Softw. Eng.*, vol. 18, no. 3, pp. 594–623, 2013.