Autonomic Metaheuristic Optimization with Application to Run-Time Software Adaptation

John M. Ewing and Daniel A. Menascé

Department of Computer Science, MS 4A5
The Volgenau School of Engineering, George Mason University
Fairfax, Virginia, United States of America
Email: jewing2@gmu.edu,menasce@cs.gmu.edu

***Abstract*—This paper presents a general meta-optimization approach for improving self-optimization in autonomic systems. This approach can improve optimization performance and lower costs by reducing human effort needed to tune optimization algorithms. We apply our meta-optimization approach to Self-Architecting Software Systems (SASSY). A genetic algorithm is used to meta-optimize both the architecture search module and the service selection search module in SASSY. Four different heuristic search algorithms (hill-climbing, beam search, evolutionary programming, and simulated annealing) are made available to be meta-optimized in both the architecture search module and the service selection search module. This meta-optimization process generated twelve new heuristic search algorithm pairs for solving SASSY optimization problems. In a large set of simulation experiments, two of the generated heuristic search algorithm pairs provided superior performance to the control (which was the previously best heuristic search algorithm pair known in SASSY).**

Keywords–Intelligent systems; Autonomous agents; Evolutionary computation; Genetic algorithms

I. INTRODUCTION

Autonomic computing is a discipline that studies the design of methods and techniques that enable information systems to manage themselves. The self-management capabilities can be broken down into four self-* properties: self-configuration, self-optimization, self-healing, and self-protection [1]. A driving force in the adaptation of autonomic computing is the desire to reduce the Total Cost of Ownership (TCO); autonomic computing achieves this goal by reducing maintenance costs, in particular the level of effort required by system administrators.

Achieving each of the self-* properties presents special challenges. In this work, we focus on the challenges presented by run-time self-optimization in the face of changes in the environment. Autonomic systems that perform self-optimization require some computational method to discover a configuration or a sequence of actions that will optimize the system. A number of techniques including linear programming, heuristic search, and machine learning have been employed to conduct self-optimization in autonomic systems [2][3][4]. Most self-optimizing autonomic systems share the following three considerations:

1) multiple optimization problems will be encountered over the life of the autonomic system,
2) encountered optimization problems must be solved in near real-time, and
3) the performance of the optimization algorithm is impacted by parameters that control the behavior of the algorithm.

For many autonomic systems, it is reasonable to expect that hundreds to thousands of optimization problems will be encountered over the system's lifetime. Self-optimization is often invoked in support of self-healing; restoring functionality to a system requires expeditious decision-making on the part of the optimizing algorithm.

Optimization conducted through heuristic search algorithms can have widely varying performance. The performance of a heuristic search algorithm largely depends upon the type of algorithm and its attendant parameter settings. The topology of the system's objective function over the system's configuration space interacts heavily with the selection of the heuristic search algorithm and attendant parameters. These interactions can be difficult to predict, and require human system administrators with significant knowledge, experience, and time to set them correctly. This additional effort can substantially reduce the original cost savings provided by the autonomic system.

To reduce costs and improve the performance of self-optimizing systems, we propose a meta-optimization technique for autonomic systems. Meta-optimization is particularly well-suited to self-optimizing autonomic systems for two reasons:

- A meta-optimized optimization algorithm is likely to yield improved results each time the algorithm is invoked. The cumulative positive impact of making better decisions over the system's lifetime can be substantial.
- Optimizations can be solved in a matter of seconds, therefore it is computationally feasible to execute the optimization algorithm thousands of times either offline or between self-optimization events.

Huebscher and McCann [5] propose classifying systems based on their degree of autonomicity. The authors suggest five levels of autonomicity:

1) *Support*–At this lowest level of autonomicity, a system focuses on only a subset self-* properties and/or focuses only on a subset of components.
2) *Core*–A system with core autonomicity enables self-* properties on all components but provides no method for modifying system goals online.
3) *Autonomous*–An autonomous system enables self-* properties on all components but does not possess awareness of the autonomic manager's performance.

 ISBN: 978-1-61208-405-3

4) *Autonomic*–An autonomic system enables self-* properties on all components, is aware of the autonomic manager's performance, and can adapt the behavior of the autonomic manager to improve performance.
5) *Closed-Loop*–A system with closed-loop autonomicity enables self-* properties on all components, is aware of the autonomic manager's performance, and grows the capabilities of the autonomic manager through intelligent reasoning.

Applying meta-optimization can contribute to the transformation of autonomous systems into autonomic systems.

This paper makes the following three contributions:

1) a framework for conducting meta-optimization on self-optimizing systems,
2) a demonstration of the framework on an application using SASSY, and
3) an experimental evaluation of the resulting meta-optimized heuristic search algorithms.

The organization of this paper is as follows. Section II provides a brief overview of the SASSY project that motivated the need for the development of the ideas presented in this paper. Section III formalizes the meta-optimization problem. The following section presents the meta-optimization framework. Section V presents and discusses the results of our experimental evaluation. The following section discusses related work and Section VII presents some concluding remarks.

II. Overview of SASSY

In previous work, we presented an autonomic framework for managing Service Oriented Architecture (SOA) applications called SASSY [6][7]. SASSY optimizes the performance of systems by modifying architectural patterns and changing service provider (SP) selections.

In SASSY, a user defines data flows among activities for a new SOA application via a graphical interface [6]. The user can specify multiple Quality of Service (QoS) requirements associated with the framework. These QoS requirements are termed service sequence scenarios (SSS) and they couple a desired QoS goal with a path through the data flows. The degree of satisfaction of the QoS goals is reflected in a global utility function, U_g, which serves as the objective function in SASSY's self-optimization processes. A detailed description of how data flows and SSSs are defined in the SASSY framework can be found in [3] and [6]. It is worth noting that the global utility functions are typically concave with multiple optima.

SASSY generates a base software architecture from the user's requirements that consists of a coordinator and a basic software component for each activity defined in the data flow. The coordinator is linked to each basic software component and SSS performance models are automatically produced using expression trees and the set of rules described in [6].

This base architecture can be modified through the substitution of a basic component with a composite component. A composite component uses multiple SPs and is created from an architectural pattern template. For example, a composite component might be constructed from a load balancing architectural pattern template; the composite component might use two different SPs and distribute the offered load according to the SPs' advertised capacities [8].

To make the architecture executable, the coordinator must bind a set of SPs to the basic components in the architecture. Different SPs may offer the same service with varying levels of performance and cost. For a given architecture, SASSY searches for a combination of SPs that maximizes U_g.

The coordinator is able to substitute patterns and components to the architecture at run-time [9]. This enables the system to re-architect at run-time when new services become available or a service currently bound to the architecture fails.

TABLE I. SSSes USED IN EXPERIMENTAL EVALUATION.

QoS Metric	Weight	Number of Activities
Security Option 1	0.08	16
Security Option 1	0.03	9
Security Option 2	0.11	11
Security Option 2	0.07	9
Throughput	0.11	11
Throughput	0.06	16
Throughput	0.02	11
Availability	0.12	16
Availability	0.08	11
Availability	0.04	16
Availability	0.04	11
Execution Time	0.18	11
Execution Time	0.03	16
Execution Time	0.03	11

Our previous work considers small- to medium-sized data flows in SASSY with up to 30 activities [3][6]. Here, we consider the much larger SOA application shown in Figure 1 that has 65 activities. A summary of the SSSes defined for this application can be found in Table I. For each SSS, the table shows its QoS metric, the weight of that metric in the computation of the global utility U_g, and the number of software components of that SSS. The heuristic search optimization algorithms considered in our previous work were tuned on an application with 30 activities. In this paper we apply a meta-optimization process to determine if more suitable heuristic search algorithms can be found for this larger application.

III. Examining Meta-Optimization

All self-optimizing systems have methods for judging the efficacy of a given configuration or sequence of actions. For the purposes of expediency in discussion, we assume that all self-optimizing systems can be gauged with a global utility function.

Formally, self-optimization can be specified as:

Find a system state S^* such that

$$S^* = \text{argmax}_S \; U_g(S, \mathcal{K}). \tag{1}$$

where $U_g()$ is a global utility function representing the usefulness of being at system state S when the operating environment is at state $\mathcal{K}$.

To achieve optimization, self-optimizing autonomic systems either employ approximate optimization algorithms or make restrictions in the number of system states that may be considered. Equation (2) shows the optimization process, B, producing an approximately optimized state, S_a^* with optimization algorithm, $\mathcal{H}$.

$$S_a^* = B(\mathcal{H}, \mathcal{K}). \tag{2}$$

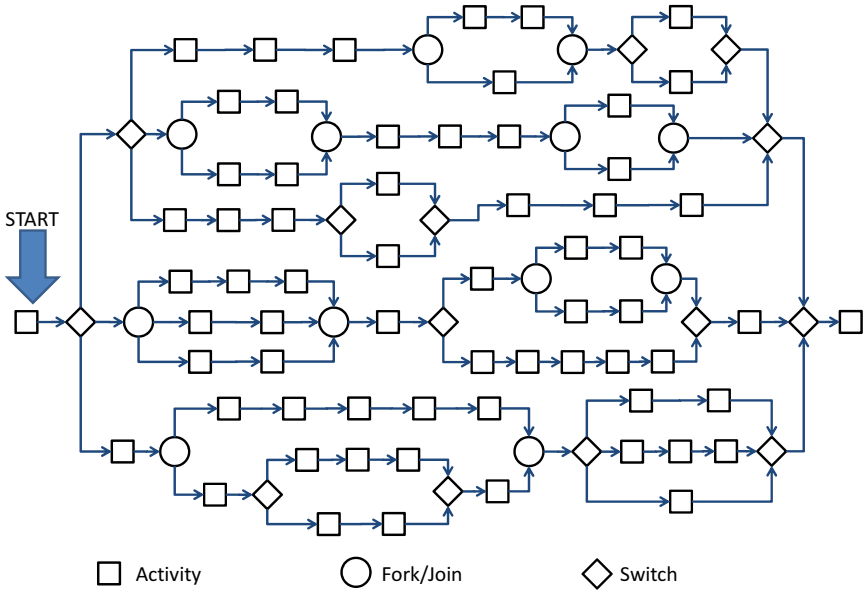

Figure 1. SOA application with 65 activities.

Often, these approximate optimization algorithms are non-deterministic due to stochastic operations (e.g., mutations in evolutionary algorithms). Thus, to measure the performance of an optimization algorithm $\mathcal{H}$, its expected global utility $\overline{U}_{\mathcal{H}}$ over multiple executions of B should be considered:

$$\overline{U}_{\mathcal{H}} = \mathbb{E}\ [U_g(S_a^*)] = \mathbb{E}\ [U_g(B(\mathcal{H}, \mathcal{K}))]. \quad (3)$$

The meta-optimization problem can be formally specified as follows:

Find an approximate optimization algorithm $\mathcal{H}^*$ such that

$$\mathcal{H}^* = \text{argmax}_{\mathcal{H}}\ \mathbb{E}\ [U_g(B(\mathcal{H}, \mathcal{K}))] \quad (4)$$

$$t_{\mathcal{H}} \leq t_L \quad (5)$$

where $t_{\mathcal{H}}$ is the execution time for $\mathcal{H}$ and t_L is a time limit.

A. Meta-Optimization in SASSY

There are two NP-hard optimization problems that need to be solved in near real-time for SASSY [6]:

1) an architecture optimization problem and
2) a service selection optimization problem.

The two optimization problems are in fact nested: before an individual architecture can be evaluated, an approximately optimal service selection must first be found.

Formally, the SASSY optimization problem can be expressed as:

Find an architecture $\mathcal{A}^*$ and a corresponding SP allocation Z^* such that

$$(\mathcal{A}^*, Z^*) = \text{argmax}_{(\mathcal{A},Z)}\ U_g(\mathcal{A}, Z, \mathcal{K}). \quad (6)$$

where $U_g(\mathcal{A}, Z)$ is the global utility of architecture $\mathcal{A}$ and service selection Z, with the state of all SPs in the environment denoted by $\mathcal{K}$. This optimization problem may be modified by adding a cost constraint. In the cost-constrained case, there is a cost associated with each SP for providing a certain QoS level [6].

The optimization process, B, used by SASSY's centralized autonomic controller requires two algorithms: $\mathcal{H}_{\mathcal{A}}$ for the architecture search and $\mathcal{H}_Z$ for the service selection search. Equation (7) shows that the optimization process requires one more input, $\mathcal{A}_c$, the current architecture. This provides a useful starting position for the algorithm $\mathcal{H}_{\mathcal{A}}$, since the $\mathcal{A}_c$ is often close to an architecture $\mathcal{A}_a^*$.

$$(\mathcal{A}_a^*, Z_a^*) = B(\mathcal{H}_{\mathcal{A}}, \mathcal{H}_Z, \mathcal{A}_c, \mathcal{K}) \quad (7)$$

The performance of the algorithm pair, $\overline{U}_{\mathcal{H}_{\mathcal{A}},\mathcal{H}_Z}$, is expressed below:

$$\begin{aligned} \overline{U}_{\mathcal{H}_{\mathcal{A}},\mathcal{H}_Z} &= \mathbb{E}\ [U_g(\mathcal{A}_a^*, Z_a^*)] \\ &= \mathbb{E}\ [U_g(B(\mathcal{H}_{\mathcal{A}}, \mathcal{H}_Z, \mathcal{A}_c, \mathcal{K}))]. \end{aligned} \quad (8)$$

Equation (9) describes the meta-optimization problem in SASSY:

Find a pair of approximate optimization algorithms $(\mathcal{H}_{\mathcal{A}}^*, \mathcal{H}_Z^*)$ such that

$$(\mathcal{H}_{\mathcal{A}}^*, \mathcal{H}_Z^*) = \text{argmax}_{(\mathcal{H}_{\mathcal{A}},\mathcal{H}_Z)}\ \mathbb{E}\ [U_g(B(\mathcal{H}_{\mathcal{A}}, \mathcal{H}_Z, \mathcal{A}_c, \mathcal{K}))] \quad (9)$$

$$t_{(\mathcal{H}_{\mathcal{A}},\mathcal{H}_Z)} \leq t_L \quad (10)$$

SASSY can employ a number of heuristic search methods as approximate optimization algorithms in solving the architectural pattern problem and the SP selection problem. Hill-climbing, beam search, simulated annealing, and evolutionary programming have been implemented and tested in the SASSY autonomic controller with varying degrees of effectiveness [3]. Each of these heuristic search algorithms requires multiple parameter settings that can have potentially large impacts on the optimization process performance.

 ISBN: 978-1-61208-405-3

IV. META-OPTIMIZATION FRAMEWORK

Meta-Optimization in SASSY is currently an offline activity that requires some minimal supervision from a human administrator.

As demonstrated in (4) and (9), certain inputs are required in the meta-optimization process. In the general case, we require the operating environment state, $\mathcal{K}$, to conduct a meta-optimization. For SASSY meta-optimizations, we additionally require the system's current architecture, $\mathcal{A}_c$.

To ensure acquisition of appropriate meta-optimization inputs, we propose the following three-step meta-optimization process:

1) capture candidate sample problem set,
2) select finalist problems from candidate problem set, and
3) apply meta-optimization procedure to finalist problems.

A candidate sample problem set is a pool of observed or generated optimization problems. A candidate sample problem set may be large, and it may not be computationally feasible to conduct effective meta-optimization on each problem in this set. When the candidate problem set is large, a method is required for selecting a promising subset (i.e., the finalists) of the candidate problems. A meta-optimization procedure can then be pursued on the small set of finalist problems.

A. Generating Candidate Problems in SASSY

To capture a candidate sample problem set in SASSY, we execute the SASSY system in a simulated service environment. The simulation generates SP failures, SP degradations, and SP repairs. If an SP failure or SP degradation reduces U_g below some threshold, the autonomic controller will initiate an optimization process to find a new architecture, $\mathcal{A}$, and SP selection, Z. When the performance monitor detects SP repair events, the autonomic controller will also initiate an optimization process to determine if a better $\mathcal{A}$ and Z can be achieved. The candidate problem set is produced by collecting randomly sampled problems encountered in the simulation—the purpose is to avoid oversampling small portions of the problem space.

In the SASSY application depicted in Figure 1, we randomly generated between three and ten SPs for each of the 65 activities, yielding an overall total of 404 SPs. We conducted a relatively long initial optimization search to find a near-optimal starting architecture, $\mathcal{A}_i$, and a near-optimal SP selection, Z_i. We instantiated a SASSY system using the beam search/evolutionary programming `BS-EP` heuristic search algorithm pair from [3]. Starting the SASSY system with $\mathcal{A}_i$ and Z_i, we simulated SP failures, SP degradation, and SP repair events over time. We conducted 26 such simulations and captured 1% of the encountered optimization problems by the SASSY autonomic controller. This process generated 1,041 candidate sample problems.

B. Selecting Finalist Problems in SASSY

Our previous work [3] demonstrated that sometimes a small fraction of SASSY optimization problems are particularly challenging. The choice of heuristic search algorithms on these challenge problems can have an outsized impact on the SASSY system's overall performance. Identifying challenge problems with machine learning techniques has proven difficult [3]. To improve the odds of including one or more challenge problems in the finalist subset, we prioritize diversity when choosing finalists from candidates.

To develop a diverse finalist subset, we examine two summary statistics:

1) $\Delta\ U_g$ is the difference in U_g from the last optimization search. This measures the severity of the optimization problem.
2) $f_\Delta(\mathcal{K})$ is the fraction of SPs that have changed state due to failure, degradation, or repair since the last optimization search. This measures the degree of change in the environment.

Figure 2 shows a scatter plot of the 1,041 candidate problems using these summary statistics.

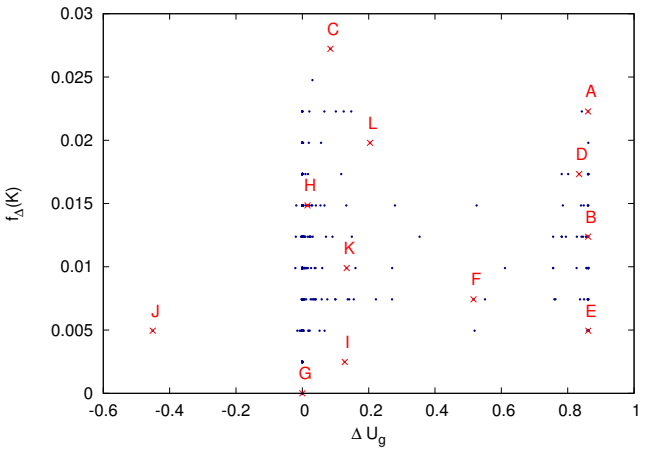

Figure 2. The candidate problem set plotted using summary statistics. The twelve finalist problems are labeled A-L and marked with red x's.

To pick a diverse group of finalist problems, we select problems distributed across the full range, including some outliers. Challenge problems may be uncommon, so it is not necessary that each finalist problem represent a cluster of candidate problems. The twelve finalist problems were selected by assessing Figure 2 and are labeled A through L.

C. Applying Meta-Optimization Procedure

Figure 3 describes the meta-optimization procedure we applied to the SASSY autonomic controller. Exactly one finalist sample problem is assigned to an instance of the meta-optimizer. The arrows departing from Box 1 show how the information captured in the finalist sample problem is distributed.

- The current architecture, $\mathcal{A}_c$, is sent to the Meta-Optimizer.
- The performance of the SPs in the environment is sent to the SSS Performance Modeler.
- A list of the available SPs in the environment is provided to the Service Selection Search Module.

The Meta-Optimizer (Box 2) generates a pair of heuristic search algorithms that are then provided to the Architecture Search Module (Box 3) and the Service Selection Search Module (Box 4). Additionally, the Meta-Optimizer directs the Architecture Search Module to commence an optimization

 ISBN: 978-1-61208-405-3

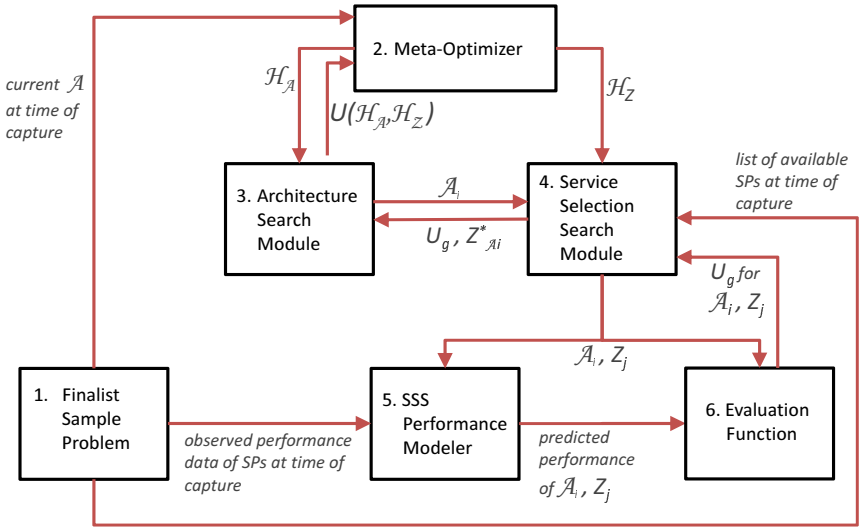

Figure 3. The meta-optimization procedure applied to SASSY.

search. The optimization search will be repeated n times before the Meta-Optimizer changes the heuristic search algorithms in the search modules (Boxes 3 and 4). The score for the heuristic search pair is the average predicted U_g of the returned $\mathcal{A}$ and Z.

The heart of the architecture/SP selection optimization is the interaction among boxes 3, 4, 5, and 6. When the architecture optimization search begins, the Architecture Search Module (Box 3) requests the Service Selection Search Module (Box 4) to find an optimal Z_i for a given $\mathcal{A}_i$. As it conducts the SP selection search, the Service Selection Search Module requests performance predictions for a given $\mathcal{A}_i$ and Z_j.

1) Genetic Algorithm for the Meta-Optimizer: We used a genetic algorithm as our meta-optimization algorithm for the following four reasons.

1) The genotype representation provides an elegant mechanism for representing complex objects.
2) The crossover and mutation operators can be applied to the genotype representation in a simple and uniform way.
3) Genetic algorithms are robust in the face of noisy evaluations.
4) The crossover operator can blend two different heuristic pair algorithms to explore the heuristic parameter space between them.

The heuristic search algorithms and their attendant parameters are encoded into binary strings. The format of these binary strings are defined in Table II and Table III. The genotype of the heuristic search algorithm pair is formed by concatenating these two binary strings. For a more detailed discussion of the heuristic search algorithm parameters, see [3].

The service selection search budget parameter, N_Z, in Table III refers to the number of SP selections to be evaluated for each architecture evaluation. Thus, the total number of model evaluations, $N_{\mathcal{M}}$ can be computed as follows:

$$N_{\mathcal{M}} = N_{\mathcal{A}} \times N_Z \tag{11}$$

where $N_{\mathcal{A}}$ is the architecture search budget parameter.

In this work, the window for completing an architecture optimization search was set to be 7.5 seconds. On systems with two 2.4 GHz quad-core hyper-threading Intel Xeon processors this translated to $N_{\mathcal{M}} = 47,600$. Using this information, $N_{\mathcal{A}}$ was then derived from N_Z.

In most genetic algorithms, the size of the parent population and offspring population are equal (in this work we use a population size of 15). With each generation, the parent solutions are discarded, and the offspring become the next generation of parents.

Each new offspring is generated by probabilistically selecting two parents. We use the linear ranking method for parent selection [10][11]. In linear ranking, the population is first sorted in descending order according to fitness, $\overline{U}_{\mathcal{H}_{\mathcal{A}},\mathcal{H}_Z}$. The probability of selecting member i is:

$$P(i) = \frac{1+\mathcal{S}}{M} - \frac{2\mathcal{S}(i-1)}{M(M-1)} \tag{12}$$

where $\mathcal{S}$ is a pressure selection variable that may take on values in the range of $[0, 1]$. When $\mathcal{S}$ is zero, all members of the population have an equal chance of being selected; as $\mathcal{S}$ increases, the probability increases of selecting the fittest members of the population. Here, we use $\mathcal{S} = 1$, which should speed the final convergence on concave maxima—this is a desirable feature given limitations on time and resources for our meta-optimization.

The offspring is produced from the two parents through the uniform crossover operator with the crossover probability set to 0.08. The genetic algorithm transcribes the binary string from the first parent selected to the offspring. With each transcribed bit, there is an 8% chance that the genetic algorithm will swap the parents for the source of the transcription [10].

Once the crossover operation is complete for a new offspring, the bit-flip mutation operator is invoked. To avoid entrapment in hamming cliffs, the binary strings are converted into Gray code [12] before the bit-flip mutation operator is applied. The bit-flip mutation operator examines each bit of the genotype binary string and flips a given bit with a probability of 0.02.

 ISBN: 978-1-61208-405-3

TABLE II. COMPOSITION OF ARCHITECTURE SEARCH BINARY STRING.

Parameter	Algorithm	Type	Min	Max	Step	# bits
search algorithm	all	enum	N/A	N/A	N/A	2
hill-climbing mode	hill-climbing	enum	N/A	N/A	N/A	1
beam search mode	beam search	enum	N/A	N/A	N/A	2
neighborhood filtering	hill-climbing & beam search	boolean	N/A	N/A	N/A	1
# of SSSes in filter	hill-climbing & beam search	integer	1	13	1	4
# of components in filter	hill-climbing & beam search	integer	1	64	1	6
beam width	beam search	integer	2	5	1	2
parent population size	evolutionary programming	integer	1	20	1	5
brood size	evolutionary programming	floating point	1.0	8.5	0.5	4
overlapping population	evolutionary programming	boolean	N/A	N/A	N/A	1
initial step size	evolutionary programming	floating point	1.0	4.5	0.5	3
adaptive step factor	evolutionary programming	floating point	1.0	4.5	0.5	3
initial probability	simulated annealing	floating point	0.1	0.7	0.04	4
final probability	simulated annealing	floating point	0.00001	0.00016	0.00001	4

TABLE III. COMPOSITION OF SERVICE SELECTION SEARCH BINARY STRING.

Parameter	Algorithm	Type	Min	Max	Step	# bits
search budget, N_Z	all	integer	100	2500	25	7
search algorithm	all	enum	N/A	N/A	N/A	2
hill-climbing mode	hill-climbing	enum	N/A	N/A	N/A	1
beam search mode	beam search	enum	N/A	N/A	N/A	2
neighborhood filtering	hill-climbing & beam search	boolean	N/A	N/A	N/A	1
# of SSSes in filter	hill-climbing & beam search	integer	1	13	1	4
# of components in filter	hill-climbing & beam search	integer	1	64	1	6
beam width	beam search	integer	2	5	1	2
parent population size	evolutionary programming	integer	1	20	1	5
brood size	evolutionary programming	floating point	1.0	8.5	0.5	4
overlapping population	evolutionary programming	boolean	N/A	N/A	N/A	1
initial step size	evolutionary programming	floating point	1.0	4.5	0.5	3
adaptive step factor	evolutionary programming	floating point	1.0	4.5	0.5	3
initial probability	simulated annealing	floating point	0.1	0.7	0.04	4
final probability	simulated annealing	floating point	0.00001	0.00016	0.00001	4

After the bit-flip mutation is complete, the genetic algorithm checks to make sure that the parameters of produced heuristic search algorithms are within acceptable boundaries. The crossover operation and bit-flip mutation are repeated as necessary to produce a valid offspring.

Each produced offspring is a pair of heuristic search algorithms for solving nested SASSY optimization problems. Each offspring is then asked to search the assigned finalist sample problem. This search is repeated n times, and the score of the heuristic pair, $\overline{U}_{\mathcal{H}_A,\mathcal{H}_Z}$, is computed as follows:

$$\overline{U}_{\mathcal{H}_A,\mathcal{H}_Z} = \frac{1}{n}\sum_{i=1}^{n} U_g(\mathcal{A}_i, Z_i) \tag{13}$$

where $\mathcal{A}_i$ and Z_i are respectively the best architecture and service selection found in optimization search instance i. In the work presented here, n has been set to 50.

The results for a given offspring are stored in a hash table. If another individual is encountered matching that offspring later in the meta-optimization search, the evaluation of the heuristic pair can be skipped, and $\overline{U}_{\mathcal{H}_A,\mathcal{H}_Z}$ can be recovered from the hash table.

The genetic algorithm continues producing new generations until the heuristic pair evaluation limit is reached (set to 1,000 evaluations in this work). This meta-optimization genetic algorithm was applied to each of the twelve finalist sample problems. The resulting heuristic pairs are shown in Table IV and Table V.

From the results in Tables IV and V, evolutionary programming is clearly the dominant heuristic search algorithm for the service selection search. At the architecture search level, a variety of local search algorithms were found to be optimal on their respective problems. A common feature across all 12 meta-optimization runs are the large values for N_Z. Only problem L generated a heuristic pair with N_Z set to less than 2,000.

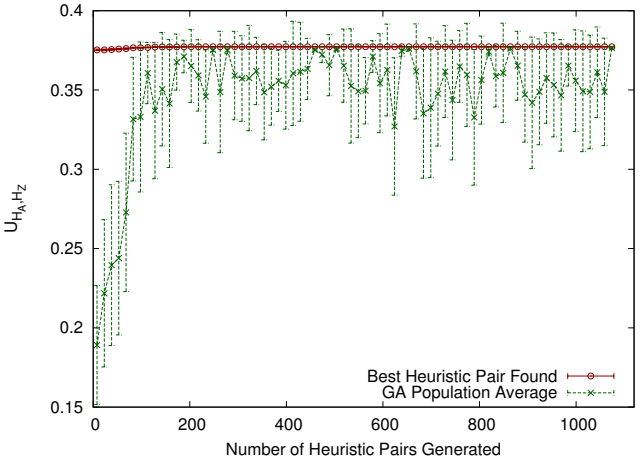

Figure 4. Heuristic pair performance on problem D with 95% CI error bars.

Figures 4 and 5 plot the progress of the meta-optimization search on the finalist sample problems D and F respectively. Due to differences in the environment, the scale of the plots' y-axis differ substantially.

 ISBN: 978-1-61208-405-3

TABLE IV. RESULTING HEURISTIC PAIRS FOR FINALIST PROBLEMS A THROUGH F.

Parameter	problem A	problem B	problem C	problem D	problem E	problem F
arch. search budget, $N_{\mathcal{A}}$	19	19	19	19	19	20
arch. search alg.	beam search	hill-climbing	hill-climbing	beam search	hill-climbing	beam search
arch. search mode	exceeds LL	greedy	opportunistic	no LL req.	greedy	no LL req.
arch. # of filter SSSes	2	6	12	4	3	4
arch. # of filter comp.	2	24	4	5	1	1
arch. beam width	4	N/A	N/A	4	N/A	5
arch. ini. prob.	N/A	N/A	N/A	N/A	N/A	N/A
arch. final prob.	N/A	N/A	N/A	N/A	N/A	N/A
serv. sel. search budget, $N_{\mathcal{Z}}$	2,475	2,475	2,475	2,475	2,475	2,275
serv. sel. search alg.	evol. prog.	evol. prog.	evo. prog.	evol. prog.	evol. prog.	evol. prog.
serv. sel. par. pop. size	2	1	1	4	1	4
serv. sel. off. pop. size	5	7	2	8	6	4
serv. sel. overlap pop.	true	true	true	true	false	true
serv. sel. ini. step size	4.5	2.5	3.0	4.5	2.5	4.5
serv. sel. adapt. step fact.	1.0	1.5	1.5	3.5	1.5	1.5

TABLE V. RESULTING HEURISTIC PAIRS FOR FINALIST PROBLEMS G THROUGH L.

Parameter	problem G	problem H	problem I	problem J	problem K	problem L
arch. search budget, $N_{\mathcal{A}}$	22	20	19	21	23	32
arch. search alg.	hill-climbing	sim. annealing	hill-climbing	hill-climbing	hill-climbing	hill-climbing
arch. search mode	opportunistic	N/A	opportunistic	greedy	opportunistic	opportunistic
arch. # of filter SSSes	11	N/A	unused	3	12	11
arch. # of filter comp.	3	N/A	unused	1	2	2
arch. beam width	N/A	N/A	N/A	N/A	N/A	N/A
arch. ini. prob.	N/A	0.26	N/A	N/A	N/A	N/A
arch. final prob.	N/A	0.0008	N/A	N/A	N/A	N/A
serv. sel. search budget, $N_{\mathcal{Z}}$	2,100	2,375	2,500	2,250	2,050	1,475
serv. sel. search alg.	evol. prog.	evol. prog.	evo. prog.	evol. prog.	evol. prog.	evol. prog.
serv. sel. par. pop. size	1	3	3	2	3	3
serv. sel. off. pop. size	4	18	22	6	12	15
serv. sel. overlap pop.	true	true	true	true	true	true
serv. sel. ini. step size	4.5	2.5	3.5	1.0	4.0	3.0
serv. sel. adapt. step fact.	1.0	1.0	2.0	1.0	1.5	1.0

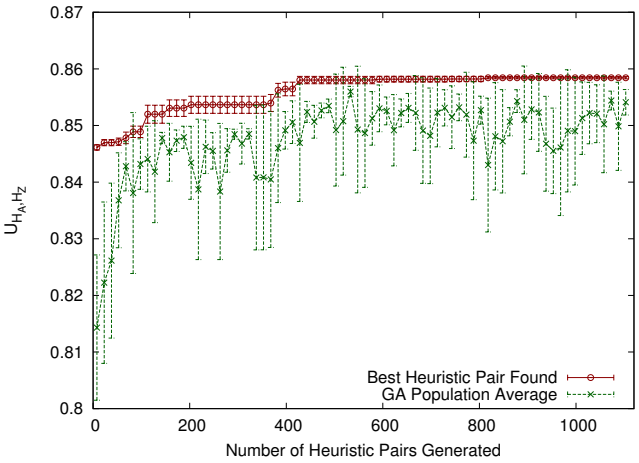

Figure 5. Heuristic pair performance on problem F with 95% CI error bars.

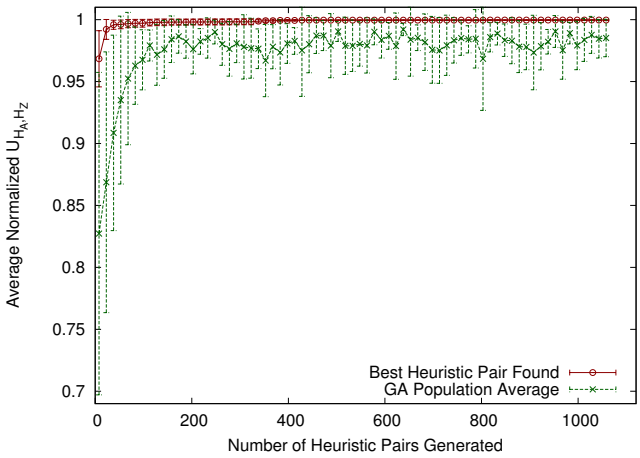

Figure 6. Normalized heuristic pair performance across all problems with 95% CI error bars.

Each of the finalist sample problems has a different y-scale. To gauge the overall convergence of the meta-optimization genetic algorithm, we normalize the search performance against the best U_g found during the entire meta-optimization search. A plot of the normalized convergence can be found in Figure 6.

V. EXPERIMENTAL EVALUATION

After the meta-optimization genetic algorithm produced optimized heuristic algorithm pairs for each of the twelve finalist problems, we tested these twelve heuristic pairs in simulation (see Tables IV and V). This simulation software was originally developed for the experimental evaluation in [3]. As a control, we also tested the beam search/evolutionary programming `BS-EP` heuristic search algorithm pair from [3] (see Table VI).

 ISBN: 978-1-61208-405-3

TABLE VI. CONTROL HEURISTIC PAIR PARAMETER SETTINGS.

Parameter	control
arch. search budget, $N_{\mathcal{A}}$	63
arch. search alg.	beam search
arch. search mode	no LL req.
arch. # of filter SSSes	5
arch. # of filter comp.	2
arch. beam width	2
arch. ini. prob.	N/A
arch. final prob.	N/A
serv. sel. search budget, $N_{\mathcal{Z}}$	756
serv. sel. search alg.	evol. prog.
serv. sel. par. pop. size	3
serv. sel. off. pop. size	19
serv. sel. overlap pop.	true
serv. sel. ini. step size	3.5
serv. sel. adapt. step fact.	4.5

A. Simulation Parameters

Each simulation commences with the SOA application in a near-optimal architecture that was found in a lengthy, offline heuristic search. The simulation time is divided into discrete intervals called *controller intervals* of duration ϵ time units.

The following actions take place at the end of each controller interval:

- SPs that are active and up will be scheduled to go down t_{fail} time units after they become operational. The time t_{fail} is drawn from an exponential distribution with an average equal to the SP's Mean Time To Failure (MTTF). This exponentially distributed number is rounded up to the closest multiple of ϵ. Thus, at the end of each controller interval, if any SP is scheduled to go down at that time, the SP is flagged as down, and the software system's U_g is computed and recorded.
- For each SP that failed at the end of a controller interval, an exponentially distributed number t_{recover} with average equal to the SP's Mean Time To Repair (MTTR) is selected. The value of t_{recover} is rounded up to the closest multiple of ϵ. Thus, at the end of a controller interval, if any SP is scheduled to recover, the SP is flagged as operational again. The autonomic controller conducts a re-architecting search to see if the new SP can be used to attain a higher U_g.
- Compute the U_g. If it falls below a certain set threshold, initiate rearchitecting.

Separate Mersenne Twister random number streams were used for the generation of simulation events and for heuristic search calculations. The duration of each simulation was 500 ϵ. We conducted 100 simulations for the control heuristic pair and for each of the twelve heuristic algorithm pairs generated by the meta-optimization process.

B. Experimental Results

Each autonomic controller encountered approximately 400 re-architecting events over the course of a single simulation run. Figure 7 shows the distribution of average global utilities in each set of 100 experiments produced by the twelve heuristic pairs and the control. The boxes in this figure show the three population quartiles, while the whiskers show the maximum and minimum.

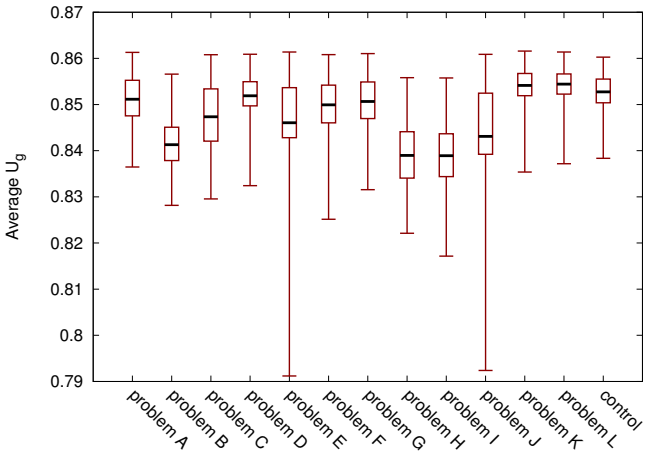

Figure 7. Box plot showing the quartiles of the simulation runs.

The average U_g maintained over the 100 simulations with 95% confidence intervals is presented in Table VII. A visual test of the confidence intervals shows that the heuristic pair generated for problem L performed better than each of the other heuristic pairs except for that generated for problem K. Next, we assess the statistical significance of the results.

TABLE VII. 95% CONFIDENCE INTERVALS FOR AVERAGE GLOBAL UTILITY.

Heuristic Pair	lower bound	mean	upper bound
control	0.8520	0.8527	0.8535
problem A	0.8501	0.8511	0.8522
problem B	0.8403	0.8413	0.8423
problem C	0.8459	0.8473	0.8488
problem D	0.8509	0.8519	0.8529
problem E	0.8436	0.8461	0.8485
problem F	0.8487	0.8499	0.8511
problem G	0.8496	0.8507	0.8518
problem H	0.8376	0.8390	0.8404
problem I	0.8376	0.8389	0.8402
problem J	0.8403	0.8431	0.8459
problem K	0.8533	0.8541	0.8550
problem L	0.8537	0.8544	0.8552

We applied the Tukey-Kramer procedure to the twelve heuristic pairs and to the control heuristic pair with $\alpha = 0.05$ and determined the following:

- The heuristic pair generated by the meta-optimization for problem L (opportunistic hill-climbing/evolutionary programming) was superior to nine of the twelve heuristic pairs generated for the other problems. Results comparing its performance to those generated for problems A, D, K, and the control were inconclusive.
- The heuristic pair generated for problem K was superior to eight of the twelve heuristic pairs generated for the other problems. Results comparing to A, D, G, L, and the control were inconclusive.
- The control pair was superior to half of the generated heuristic pairs; the results comparing to A, D, F, G, K, and L were inconclusive.

To obtain more conclusive results, we reduced the variance caused by the inferior performance of certain heuristic pairs

 ISBN: 978-1-61208-405-3

by repeating the test with the top performing heuristic pairs, thereby increasing the granularity of the Tukey-Kramer procedure. When considering just the heuristic pairs generated for problems A, D, K, L, and the control, we found the following:

- The heuristic pair generated for problem L was superior to those generated for problems A and D.
- The heuristic pair generated for problem K was superior to that generated for problem D.

We further reduced the variance to permit comparisons among the top three heuristic pairs: for problem K, for problem L, and the control. We found the following:

- The heuristic pair generated by the meta-optimization for problem L was superior to the control.
- The heuristic pair generated by the meta-optimization for problem K was also superior to the control.

VI. Related Work

Early work in meta-optimization of heuristic search algorithms was performed by Grefenstette [13]. In this work, genetic algorithms (GAs) were used to optimize other GAs. The motivation for this work was similar to ours: a reduction in the human effort required to select appropriate parameters controlling the GA's behavior. Similar to Grefenstette, Keane [14] focuses on meta-optimization of GAs used in multi-peak engineering problems. The GAs are meta-optimized by both GAs and simulated annealing. A more sophisticated approach that focuses on improving GA performance on mixed integer optimization is presented by Bäck in [15]. In this work, the meta-optimization algorithm is a hybrid of evolution strategies and a GA.

In [16], Meissner et al. develop a particle swarm optimization (PSO) meta-optimization technique using a super-particle swarm that manages the parameters of sub-particle swarms with a focus on optimizing neural networks. In his dissertation thesis [17], Pedersen presented a meta-optimization that he applied to PSO and Differential Evolution. His meta-optimizer found simpler algorithms were often more effective.

In [18], Stephenson et al. employ an evolutionary algorithm for meta-optimizing compiler heuristics. Similar to our work here, reducing human effort in tuning heuristics was a primary motivation for this work.

A literature review of software architecture optimization that provides a useful roadmap for comparing features and categorizing work in this field can be found in [19].

In [20], Calinescu et al. present QoSMOS, a system for online performance management of SOA systems. Like SASSY, this system employs utility functions to combine multiple QoS objectives and optimizes the selection of SPs. Unlike SASSY QoSMOS considers the SPs to be white boxes, and it can adjust the configuration parameters and ersource allocations for those white box SPs. Also, QosMOS does not employ architectural patterns for improving QoS. Finally, QoSMOS uses exhaustive search, a technique that cannot be used in near real-time at the scale presented in our paper.

Cardellini et al. devise a framework, MOSES, for optimizing SOA systems in [4]. Similar to SASSY, MOSES uses SP selection and architectural patterns for improving the QoS of a SOA service or application. MOSES adapts the optimization problem such that it can be solved through linear programming (LP) techniques. LP techniques operate well on convex objective functions but are substantially less effective on concave objective functions with multiple optima. The optimization techniques presented in our paper are more effective on concave global utility functions with multiple optima.

Other researchers have investigated using multi-objective optimization techniques to reduce effort and increase the quality of software architecture designs. When the optimization search completes, these systems present human decision makers with a set of Pareto optimal architecture candidates. PerOpteryx, introduced by Koziolek et al. in [21], employs architectural tactics in a multi-objective evolutionary algorithm to expedite the multi-objective search process; later work extends this approach in [22]. Martens et al. present a similar system in [23] that starts quickly by using LP on a simplified version of the problem to prepare a starting population for a multi-objective evolutionary algorithm.

VII. Conclusion

The meta-optimization was successful. Some of the resulting heuristic pairs exceeded even the performance of the control, which had previously been shown to be optimal on a different SASSY application [3], and which performed well in comparison to many of the meta-optimized heuristic pairs in these experiments.

Of the twelve heuristic pairs generated by the meta-optimization, the heuristic pairs produced for problems K and L possessed the largest architecture search budgets (23 and 32 respectively), while the control heuristic pair had an architecture search budget of 63. These settings are likely due to the more challenging nature of problems K and L as compared to A through J. Both the K and L heuristic pairs use opportunistic hill-climbing for the architecture search algorithm; this leverages the architecture search budget by ensuring the search can visit a number of architecture neighborhoods.

For this SASSY application, having an effective architecture search is key to succeeding on the more challenging optimization problems. Those heuristic pairs produced for less challenging problems de-emphasized the architecture search in favor of the service selection search. This provides marginal benefits when solving the easiest problems, but is a significant liability on more challenging problems and can lead to lower global utility values over time.

The relatively wide range in the performance of meta-optimized heuristic pairs highlights the importance of running the meta-optimization on a diverse set of problems, including outliers (both problems K and L were outliers). When performing future meta-optimizations in SASSY, we will consider using a larger set of finalist sample problems to ensure the presence of challenging problems.

Using meta-optimized heuristic pairs on SASSY provides cumulative global utility benefits over time. Furthermore, the generation of the meta-optimized heuristic pairs was automated and required minimal human administration. The meta-optimization process lowered costs by reducing the human effort required to find effective heuristic pairs. Thus, we have achieved better performance at reduced cost.

In future work, the meta-optimization process could be fully automated. This would allow online SASSY meta-controllers in [3] to use the meta-optimization framework

 ISBN: 978-1-61208-405-3

presented here. A logical question when considering meta-optimization is: "What or who will manage the meta-optimization process?" Like the autonomic controller it manages, the meta-controller contains a number of tunable parameters. Has the introduction of the meta-optimization process moved the management overhead to a new component?

Although setting up a meta-optimization process requires some initial effort from human administrators, there is an argument that this effort will be minimal compared to managing the autonomic controller itself. The autonomic controller is closer to the dynamic environment of the managed system than the meta-optimization process. This dynamism can cause problems for an autonomic controller.

However, the immediate environment of the meta-optimization process is more static. The meta-optimization's environment changes only when large changes are made to the autonomic controller (e.g., the introduction of new heuristic search algorithms or a significant evolution of the managed SOA application). Even when such large changes occur, a properly constructed and tested meta-optimization process should be able to weather the change with minimal human intervention. Thus, the meta-optimization process represents a significant step towards developing fully *autonomic* systems.

Finally, we believe the overall meta-optimization approach presented here could be adopted in other self-adaptive, self-optimizing autonomic systems.

REFERENCES

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," IEEE Computer, vol. 36, no. 1, Jan. 2003, pp. 41–50.

[2] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A hybrid reinforcement learning approach to autonomic resource allocation," in Proc. 3rd IEEE International Conference on Autonomic Computing (ICAC '06), Dublin, Ireland, Jun. 2006, pp. 65–73.

[3] J. M. Ewing and D. A. Menascé, "A meta-controller method for improving run-time self-architecting in soa systems," in Proceedings of the 5th ACM/SPEC international conference on Performance engineering. ACM, 2014, pp. 173–184.

[4] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, and R. Mirandola, "Moses: A framework for QoS driven runtime adaptation of service-oriented systems," Software Engineering, IEEE Transactions on, vol. 38, no. 5, 2012, pp. 1138–1159.

[5] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing–degrees, models, and applications," ACM Computing Surveys, vol. 40, no. 3, Aug. 2008, pp. 1–28.

[6] 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 Workshop on Software and Performance, San Jose, CA, Jan. 2010, pp. 27–36.

[7] D. A. Menascé, H. Gomaa, S. Malek, and J. Sousa, "Sassy: A framework for self-architecting service-oriented systems," IEEE Software, vol. 28, no. 6, Nov. 2011, pp. 78–85.

[8] D. A. Menascé, J. P. Sousa, S. Malek, and H. Gomaa, "QoS architectural patterns for self-architecting software systems," in Proc. 7th International Conference on Autonomic Computing (ICAC '10), Washington, DC, Jun. 2010, pp. 195–204.

[9] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, and D. A. Menascé, "Software adaptation patterns for service-oriented architectures," in Proc. 2010 ACM Symposium on Applied Computing, Sierre, Switzerland, Mar. 2010, pp. 462–469.

[10] K. DeJong, Evolutionary Computation. Cambridge, MA: MIT, 2002.

[11] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, Eds., Modern Heuristic Search Methods. Hoboken, NJ: Wiley, 1996.

[12] J. Rowe, D. Whitley, L. Barbulescu, and J.-P. Watson, "Properties of gray and binary representations," Evolutionary Computation, vol. 12, no. 1, 2004, pp. 47–76.

[13] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms," Systems, Man and Cybernetics, IEEE Transactions on, vol. 16, no. 1, 1986, pp. 122–128.

[14] A. J. Keane, "Genetic algorithm optimization of multi-peak problems: studies in convergence and robustness," Artificial Intelligence in Engineering, vol. 9, no. 2, 1995, pp. 75–83.

[15] T. Bäck, "Parallel optimization of evolutionary algorithms," in Parallel Problem Solving from NaturePPSN III. Springer, 1994, pp. 418–427.

[16] M. Meissner, M. Schmuker, and G. Schneider, "Optimized particle swarm optimization (opso) and its application to artificial neural network training," BMC bioinformatics, vol. 7, no. 1, 2006, p. 125.

[17] M. E. H. Pedersen, "Tuning & simplifying heuristical optimization," Ph.D. dissertation, University of Southampton, 2010.

[18] M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," SIGPLAN Not., vol. 38, no. 5, 2003, pp. 77–90.

[19] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," Software Engineering, IEEE Transactions on, vol. 39, no. 5, 2013, pp. 658–683.

[20] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic QoS management and optimization in service-based systems," Software Engineering, IEEE Transactions on, vol. 37, no. 3, 2011, pp. 387–409.

[21] A. Koziolek, H. Koziolek, and R. Reussner, "Peropteryx: automated application of tactics in multi-objective software architecture optimization," in QoSA-ISARCS '11. New York, NY, USA: ACM, 2011, pp. 33–42.

[22] A. Koziolek, D. Ardagna, and R. Mirandola, "Hybrid multi-attribute qos optimization in component based software systems," Journal of Systems and Software, vol. 86, no. 10, 2013, pp. 2542–2558.

[23] 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 Research into Practice–Reality and Gaps, ser. LNCS, G. Heineman, J. Kofron, and F. Plasil, Eds. Springer Berlin Heidelberg, 2010, vol. 6093, pp. 84–101.

 ISBN: 978-1-61208-405-3