A Meta-Controller Method for Improving Run-Time Self-Architecting in SOA Systems

John M. Ewing and Daniel A. Menascé
Department of Computer Science, MS 4A5
Volgenau School of Engineering
George Mason University
4400 University Dr., Fairfax, VA 22030
{jewing2,menasce}@gmu.edu

ABSTRACT

This paper builds on SASSY, a system for automatically generating SOA software architectures that optimize a given utility function of multiple QoS metrics. In SASSY, SOA software systems are automatically re-architected when services fail or degrade. Optimizing both architecture and service provider selection presents a pair of nested NP-hard problems. Here we adapt hill-climbing, beam search, simulated annealing, and evolutionary programming to both architecture optimization and service provider selection. Each of these techniques has several parameters that influence their efficiency. We introduce in this paper a meta-controller that automates the run-time selection of heuristic search techniques and their parameters. We examine two different meta-controller implementations that each use online learning. The first implementation identifies the best heuristic search combination from various prepared combinations. The second implementation analyzes the current self-architecting problem (e.g. changes in performance metrics, service degradations/failures) and looks for similar, previously encountered re-architecting problems to find an effective heuristic search combination for the current problem. A large set of experiments demonstrates the effectiveness of the first meta-controller implementation and indicates opportunities for improving the second meta-controller implementation.

Categories and Subject Descriptors

G.1.6 [**Optimization**]: Global optimization, Simulated annealing; I.2.8 [**Artificial Intelligence**]: Problem solving, Control methods, and Search; D.2.11 [**Software Architectures**]: Patterns; D.4.8 [**Performance**]: Stochastic analysis; C.4 [**Modeling Techniques**]: Experimentation

Keywords

automated run-time software architecting, autonomic computing, meta-controlled QoS optimization, combinatorial search techniques, heuristic search, metaheuristics, SOA

ICPE'14, March 22–26, 2014, Dublin, Ireland.

http://dx.doi.org/10.1145/2568088.2568098.

1. INTRODUCTION

Service Oriented Architectures (SOA) present many interesting potential benefits and challenges [10]. SOA software systems can be composed on the fly through service discovery. We assume an environment in which there are many functionally equivalent service providers (SPs) that may exhibit different quality of service (QoS) attributes at different cost. In recent work, the authors of this paper and colleagues at George Mason University developed a framework called Self-Architecting Software Systems (SASSY) [26, 27] that allows domain experts to specify the requirements of an SOA software system using a visual activity-based language [11]. SASSY automatically finds a software architecture and a selection of SPs that maximizes a given utility function of various QoS attributes subject to cost constraints. SASSY monitors the operation of the system at run-time and performs adaptation by re-architecting and selecting new SPs as needed. The general adaptation performed by SASSY follows the MAPE-K autonomic model [17] as well as the Kramer and McGee three-layer adaptation model [20].

Many autonomic controllers are based on search methods supported by performance models [15]. The models enable prediction of the performance of any potential system configuration [1, 12, 25, 26]. Then, a search algorithm is employed to explore the system configuration space in a quest for the most suitable system configuration. Typically, the most suitable configuration is the one that maximizes utility. The search algorithms employed range from simple exhaustive search to complex heuristics. Heuristic search supported by modeling has proven robust even in certain cases where the assumptions of the performance model do not hold [1].

Parameter tuning was an early application of adaptive systems employing heuristic search [25], which has proven to be a popular choice for resource allocation [12, 16, 29, 31]. Recent work with these methods has focused on selecting optimal software architectures [26] and service selections [4, 7, 23, 24] at run-time.

An excellent roadmap that summarized the state-of-the-art and identified critical challenges for the systematic software engineering of self-adaptive systems was presented by Cheng et. al. [5]. The approach to self-adapting software systems presented here is based on software architectures, i.e., it is a white box approach. A different approach, which falls in the category of black box approaches, is based on adaptation by selectively enabling and disabling software features. An example of this approach is the FeatUre-oriented

Self-adaptatION (FUSION) framework, which learns the impact of adaptation decisions on the system's goals [8, 9].

This paper builds on SASSY, which automatically re-architects an SOA software system when services fail or degrade. Optimizing both architecture and SP selection presents a pair of nested NP-hard problems. Here we significantly extend SASSY in two ways: (1) we adapt hill-climbing, beam search, simulated annealing, and evolutionary programming to both architecture optimization and SP selection; and (2) we introduce a meta-controller to automate the run-time selection of heuristic search techniques and their parameters.

We examine two different meta-controller implementations that each use online learning. The first implementation identifies the best heuristic search combination from various prepared combinations. The second implementation analyzes the current self-architecting problem (e.g. changes in performance metrics, service degradations/failures) and looks for similar, previously encountered re-architecting problems to find an effective heuristic search combination for the current problem. A large set of experiments demonstrates the effectiveness of the first meta-controller implementation and indicates opportunities for improving the second meta-controller implementation.

The rest of this paper is organized as follows. Section two presents definitions used in the paper. Section three defines the optimization problem to be solved by the self-adapting self-architecting framework. The next section discusses the framework and all its modules. Section five presents four heuristic algorithms and the challenges of heuristic search. Section six presents two different approaches to the meta-controller. The next section presents and discusses the experimental results. Section eight considers the related work. Finally, section nine presents concluding remarks.

2. DEFINITIONS

This section defines the terms used in this paper. These definitions are not meant to define a new software architectural description language [15] but to establish the concepts required in the paper at a sufficient level of abstraction.

Definition 1 (basic software component): a piece of software that has a well-defined interface that specifies the functions performed by the component. A software component can be composed with other components, can be reused, and independently implements its functions.

Definition 2 (composite software component): an atomic composition of components (basic or composite) that has an interface equivalent to a basic software component. The interface of a composite component is called a *connector*.

Definition 3 (link): a tuple (v, w) where v and w are either basic or composite software components and v invokes a function provided by w.

Definition 4 (software architecture, $\mathcal{A}$): the tuple $(\mathcal{C}, \mathcal{L}, \mathcal{S})$ where $\mathcal{C}$ is a set of basic or composite software components, $\mathcal{L} = \{(v, w) \mid v, w \in \mathcal{C}\}$ is a set of links, and $\mathcal{S}$ is a set of service sequence scenarios defined below.

Definition 5 (service sequence scenario, SSS): an SSS of the software architecture, $\mathcal{A}$, is the tuple $(\Theta, q, U(q))$ where (1) $\Theta = (\mathcal{C}_s, \mathcal{L}_s)$ is such that $\mathcal{C}_s \subseteq \mathcal{C}, \mathcal{L}_s \subseteq \mathcal{L}$, and $\forall (v, w) \in \mathcal{L}_s, v, w \in \mathcal{C}_s$; (2) q is a QoS metric, and (3) $U(q)$ is an attribute utility function, discussed below, of metric q.

Figure 1 provides a pictorial example of a software architecture, $\mathcal{A}$, where $\mathcal{C} = \{C_1, C_2, C_3, C_4, C_5\}$, $\mathcal{L} = \{L_1, L_2, L_3, L_4, L_5\}$, and $\mathcal{S} = ((\mathcal{C}_s, \mathcal{L}_s), r, U(r))$ where $\mathcal{C}_s = \{C_1, C_2, C_3\}$, $\mathcal{L}_s = \{L_1, L_3\}$, r is the response time metric, and $U(r)$ is a utility function of r.

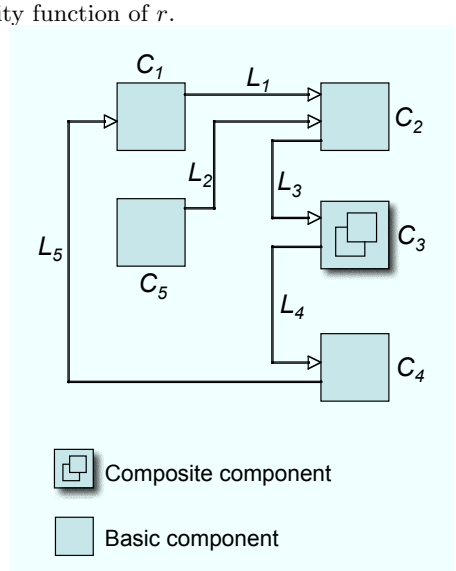

Figure 1: Depiction of an architecture.

Definition 6 (SOA software system): the result of instantiating a software architecture $\mathcal{A} = (\mathcal{C}, \mathcal{L}, \mathcal{S})$ in which the basic software components, including those that are part of composite software components, in $\mathcal{C}$ are instantiated by SPs available in an SOA environment. The selection of SPs to instantiate the basic software components of an architecture is denoted by Z.

Definition 7 (attribute utility function, $U(q)$): a function that maps a value of q to a number $u \in [0, 1]$ in a way that larger values of u correspond to better values of q. A performance model for q can be used to predict q from a given $\mathcal{A}$ and Z.

Definition 8 (global utility function, $U_g(U_1(q_1), \cdots, U_m(q_m))$): a function of the attribute utility functions of all the SSSes. The value of the function U_g must $\in [0, 1]$.

Definition 9 (SSS performance model, $\mathbb{E}(q)$): a performance model for the SSS $(\Theta, q, U(q))$ that is a function (or algorithm) used to compute the value of the performance metric q for the SSS.

3. THE OPTIMIZATION PROBLEM

In SASSY, a domain expert describes data flows between activities for a new SOA application in a visual language [26]. The domain expert can specify multiple QoS requirements which are then expressed as SSSes and attribute utility functions defined in the previous section. SSSes and attribute utility functions can also be used to specify different security options and the utility payoff for achieving specific levels of security on each component in the SSS. The domain expert then specifies a global utility function that combines the attribute utility functions.

When these requirements are finalized, SASSY generates a base software architecture that consists of a coordinator and a basic software component for each activity described in the data flow. Each basic software component is linked

to the coordinator, and SSS performance models are automatically generated using an expression tree and the set of rules described in [26].

More sophisticated architectures can be derived from the base architecture by substituting composite components for basic components. Specific architectural patterns can be used as templates for composite components. SASSY employs load-balancing and fault-tolerant architectural patterns to improve the QoS in the specified SSSes [28]. SASSY seeks to find an architecture that can provide the greatest U_g.

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 [14]. This enables the system to re-architect at run-time when new services become available or a service currently bound to the architecture fails.

The self-architecting optimization problem is to find the software architecture $\mathcal{A}^*$ and the SP selection Z^* such that U_g is optimized. More formally, the optimization problem can be expressed as:

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

$$(\mathcal{A}^*, Z^*) = \text{argmax}_{(\mathcal{A},Z)}\ U_g(\mathcal{A}, Z). \quad (1)$$

$U_g(\mathcal{A}, Z)$ is the global utility function of architecture $\mathcal{A}$ and service selection Z.

This optimization problem may be modified by adding a cost constraint. In the cost-constrained case, one assumes that there is a cost associated with each SP for providing a certain QoS level [26].

The number of different architectures is $O(p^n)$ where p is the average number of architectural patterns that can be used to replace any component and n is the number of components in the architecture. The number of possible SP selections for an architecture with n components is $O(s^n)$ where s is the average number of SPs that can be used to implement each component. Thus, the size of the solution space for this optimization problem is $O((p \times s)^n)$. The solution space is huge even for small values of p, s, and n; in fact, the problem is NP-hard. For example, for $p = 5$, $s = 2$, and $n = 10$, the size of the solution space is on the order of 10^{10}, i.e., 10 billion possible solutions [26].

Without an accompanying service selection, Z, performance models cannot predict the performance of an architecture, $\mathcal{A}$. Thus, each evaluation of an architecture $\mathcal{A}$ requires a new NP-hard search of the service selection space for the service allocation Z that maximizes the U_g of $\mathcal{A}$.

A search for a near-optimal architecture $\mathcal{A}$ requires a sequence of transformations from one architecture to another. We restrict these transformations to those that replace the initial basic components with functionally equivalent architectural patterns, i.e., composite components. More precisely, the only allowed transformations from an architecture $\mathcal{A}_i = (\mathcal{C}_i, \mathcal{L}_i, \mathcal{S}_i)$ into a different architecture $\mathcal{A}_j = (\mathcal{C}_j, \mathcal{L}_j, \mathcal{S}_j)$ are those that replace a basic component $c \in \mathcal{C}_i$ with a composite component c' or replace a composite component $c' \in \mathcal{C}_i$ with a basic component c. The replacement of components is driven by the goal to optimize U_g of the architecture. We select composite components from a library of QoS architectural patterns [28]. We also consider changes to the security level to be architecture transformations rather than changes to the selection of SPs.

The presence of nested NP-hard optimization problems in the software architecture optimization problem suggests the need for effective heuristic search. The optimal selection of SPs for a given architecture is similar to the problem of optimal service allocation for business processes in SOAs described in [7].

Most research on NP-hard optimization problems has focused on local search algorithms and evolutionary algorithms [30]. It should be noted that local search algorithms and evolutionary algorithms are not guaranteed to find the global optimum; however in most cases they find near-optimal solutions. Sacrificing an optimal solution for a near-optimal solution is usually an acceptable tradeoff to avoid a costly exhaustive search of the exponentially-sized solution spaces found in NP-hard problems.

For autonomic computing systems, the time and resources available for heuristic search may be substantially limited. Often, an optimization search will be spurred by changes in the autonomic controller's environment, and the controller will need to respond to these changes within a matter of seconds. Therefore, good heuristic search performance is essential for the autonomic controller.

Two particular characteristics are of concern in heuristic search performance: 1) the ability to avoid entrapment in local optima and 2) the convergence rate. The convergence rate measures improvement in the best predicted U_g with respect to either the number of evaluations or processing time consumed by the search.

For the autonomic controller presented here, the utility landscapes of the configuration spaces may vary widely due to differences in utility functions, application designs, and environments. The behaviors of heuristic search algorithms vary considerably and often interact with the ruggedness of the utility landscape. All search heuristics seek to balance *exploration* of previously unvisited portions of the search space with *exploitation* of promising areas of the search space. On smoother utility landscapes, exploitative heuristic search algorithms are likely to experience higher convergence rates than exploration-oriented algorithms. On rougher utility landscapes, exploration-oriented algorithms are more likely to avoid entrapment in local optima than exploitative algorithms.

4. ARCHITECTURAL SELF-ADAPTATION FRAMEWORK

This section describes our framework for self-adaptation and architecture/component selection optimization. Figure 2 shows the modules and data flows in the proposed monitoring and optimization framework. An architecture optimization search is started when either:

- the performance monitor (box 1) detects that a decline in U_g has crossed some threshold or
- the service registry (box 7) notifies the meta-controller that a new SP has become available.

The performance monitor sends a message to the meta-controller (box 2). The meta-controller selects an appropri-

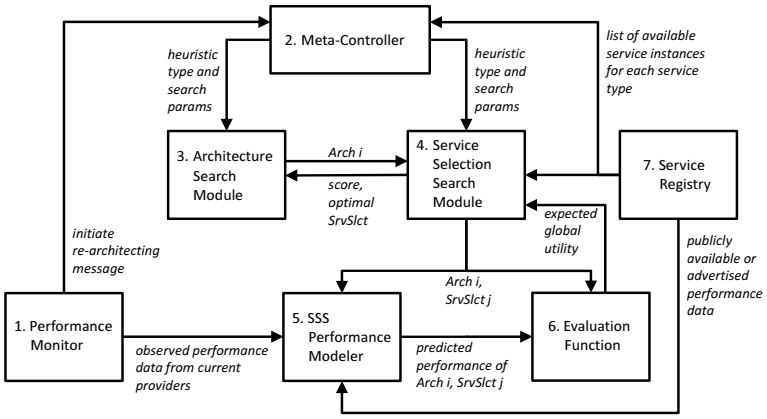

Figure 2: Data flows in the meta-controller monitoring and optimization framework.

ate heuristic search procedure, $\mathcal{H}_{Arch}$ for the architecture search module (box 3) and a potentially different heuristic search procedure, $\mathcal{H}_{SrvSlct}$, for the service selection search module (box 4).

The architecture search module (box 3) commences the execution of the heuristic search procedure, $\mathcal{H}_{Arch}$. Whenever $\mathcal{H}_{Arch}$ requests an evaluation (i.e., prediction of U_g) for a specified architecture, $\mathcal{A}_i$, the architecture search module (box 3) passes $\mathcal{A}_i$ to the service selection search module (box 4).

At this point, the service selection search module (box 4) initiates a new search that takes $\mathcal{A}_i$ as input and executes the heuristic search procedure $\mathcal{H}_{SrvSlct}$. Whenever $\mathcal{H}_{SrvSlct}$ requests an evaluation of service selection, Z_j, the service selection search module (box 4) passes a copy of $\mathcal{A}_i$ and Z_j to each of the SSS performance modeler (box 5) and the evaluation function (box 6).

The SSS performance modeler (box 5) predicts the QoS metrics for each SSS and passes the results to the evaluation function (box 6). The evaluation function applies the attribute utility functions of each SSS to the QoS metrics. The resulting SSS utility values are fed into the U_g function.

The evaluation function (box 6) returns $U_g(\mathcal{A}_i, Z_j)$ to the service selection search module (box 4), and $\mathcal{H}_{SrvSlct}$ uses this as the fitness score for Z_j and continues the search. The heuristic $\mathcal{H}_{SrvSlct}$ persists searching until some exit criterion is met (e.g. threshold utility is achieved or evaluation budget consumed). When $\mathcal{H}_{SrvSlct}$ completes, the service selection search module (box 4) returns $U_g^{\text{best}}(\mathcal{A}_i, Z_{best})$ and Z_{best} to the architecture search module (box 3).

With the completion of a service selection search instance, $\mathcal{H}_{Arch}$ uses $U_g^{\text{best}}(\mathcal{A}_i, Z_{best})$ as the fitness score for $\mathcal{A}_i$ and continues the search. The heuristic $\mathcal{H}_{Arch}$ persists searching until some exit criterion is met (e.g. threshold utility is achieved or evaluation budget consumed). When $\mathcal{H}_{Arch}$ completes, the architecture search module (box 3) sends $\mathcal{A}_{best}$ and Z_{best} to the change planner/manager (not shown in Fig. 2). The change planner/manager then executes a plan for online evolution or adapation of the running system.

This framework assumes the existence of a service registry (box 7) that includes QoS levels of the service instances listed in the registry [6]. This information is required by three modules: the meta-controller (box 2) that uses this information when selecting $\mathcal{H}_{Arch}$ and $\mathcal{H}_{SrvSlct}$, the service selection search module (box 4) that needs to know which SPs are available for the search, and the SSS performance modeler (box 5) that uses the advertised performance of the SPs.

The performance monitor (box 1) continuously collects QoS metrics and tracks U_g in real-time. As mentioned at the start of this section, the performance monitor (box 1) can initiate a new architecture search if U_g (most likely represented as a moving average) declines below a threshold utility level that was set upon completion of the last architecture search. The performance monitor (box 1) continuously sends performance data updates to the SSS performance modeler (box 5), which stores near term performance data so that it is prepared to support optimization searches.

5. HEURISTIC ALGORITHMS EMPLOYED

This section describes one of the main contributions of this paper, i.e., the adaptation of well-known heuristic algorithms to architecture search and service selection. In particular, we have adapted the following heuristic algorithms: hill-climbing, beam search, simulated annealing, and evolutionary programming. Hill-climbing, beam search, and simulated annealing belong to the local search family of heuristic algorithms. Local heuristic search algorithms (known as direct search in the operations research community [18]) start with one or more solutions (referred to as the *visited solutions*) and then evaluate similar solutions called neighbors. In an effort to find better solutions, a local search algorithm will then visit one or more promising neighbor solutions and generate new neighborhoods to evaluate from those visited solutions. The search proceeds until either the search budget has been exhausted or a local optimum has been found. Most local search algorithms, after identifying a local optimum, will restart the search from a randomly selected solution(s) in an attempt to locate a better optimum.

5.1 Hill-Climbing

Hill-climbing is a relatively simple local search method that visits only one solution at a time. Hill-climbing can operate in either a greedy mode or an opportunistic mode. A greedy hill-climber evaluates an entire neighborhood before visiting one of the neighboring solutions. The greedy hill-climber will visit the highest utility solution in the neighborhood so long as that solution offers a utility improvement over the currently visited solution. The greedy hill-climber then generates a new neighborhood when it visits this new

neighbor. An opportunistic hill-climber evaluates members of the neighborhood one at a time in a randomly selected order. If any neighbor offers an improvement in score over the currently visited solution, the opportunistic hill-climber will move to visit that solution and generate a new neighborhood, neglecting the evaluation of the rest of the former neighborhood. In either mode, when the hill-climber becomes stuck in local optima, it may select a random solution and recommence the search.

5.2 Beam Search

Beam search is similar to hill-climbing but visits multiple solutions at the same time. The currently visited solutions in beam search are referred to as the *level-list*. The maximum size of the level-list is called the *beam width*. The beam search algorithm generates neighbors for each member of the level-list. The best solutions from the combined neighborhood are then selected for the next level-list. Optional selection requirements may also be applied to the new level-list. Our implementation of beam search stores previously used level-list solutions in a hash table, so that no solution makes more than one appearance on the level-list. This allows beam search to move down the utility landscape and potentially out of a local optimum.

5.3 Neighborhood Filtering (Hill-Climbing and Beam Search)

The definition of the neighboring solutions is a key to the success of local search heuristic algorithms. For configuration optimization problems, local search typically will define the neighborhood as any configuration that has a single change from the currently visited solution. For many medium to large configuration optimization problems, such a neighborhood definition could lead to large, unwieldy neighborhoods that reduce the effectiveness of the search.

In our previous work [26], we applied heuristic filtering to reduce the size of the neighborhood. A neighborhood heuristic filter examines the shortcomings of the currently visited solution and identifies and visits only those neighboring solutions that are most likely to have an improved U_g score.

Filtered neighborhood construction in architecture search attempts to improve U_g by addressing the k SSSs with the largest negative impact on U_g. In some of the neighborhood generation rules, only candidate components are considered for modification. The j worst performing components for the given SSS metric are designated as the candidate components.

For non-security SSSs, neighbors are produced in the following ways:

- For each of the j candidate components: 1) neighbors are produced by substituting architectural patterns [28] that are expected to improve the metric of the SSS and 2) neighbors are produced by incrementing/decrementing the number of service instances in that component.

- If the non-security SSS has a common component with a security SSS, a neighbor is produced by decrementing the security option level along the entire path of the security SSS.

If the SSS is a security SSS (i.e., the SSS metric is a security option), then a neighbor is produced by incrementing the level of that option along the entire SSS path.

Neighborhood construction in service selection search also considers the k non-security SSSs with the largest negative impact on U_g. For each of the k SSSs and for each of their j candidate components, the lowest performing service instance is identified according to the SSS metric. If an unused service instance offers a performance improvement in the SSS metric, a neighbor is produced by substituting in the superior service instance.

5.4 Simulated Annealing

Simulated annealing is a stochastic local search heuristic that operates like opportunistic hill-climbing with one key difference: simulated annealing may stochastically decide to visit inferior (i.e., lower predicted U_g) neighbors [30]. The probability of visiting an inferior neighbor i is determined as follows [30]:

$$p\left(V_i^{inf}\right) = e^{\left(\frac{-\Delta U_g^i}{T}\right)} \tag{2}$$

where ΔU_g^i is the difference in global utilities between the currently visited solution and neighbor i, and T is the temperature variable. When T is large, the probability that simulated annealing will decide to visit a significantly inferior neighbor is high. When T is small, simulated annealing is less likely to visit inferior neighbors and its behavior will start to resemble a deterministic hill-climber. To simulate the cooling process, T is gradually reduced as the search proceeds. The process by which T is reduced is referred to as the cooling schedule. In this work, we employ an exponential cooling schedule:

$$T_{i+1} \leftarrow \alpha T_i \tag{3}$$

where α is a constant between 0 and 1, and i is the number of completed evaluations. An accepted rule of thumb for determining the initial temperature, T_0, is to ensure at the start of the search a roughly 40% to 60% chance that a significantly inferior neighbor will be visited. When using an exponential cooling schedule, T_0 can be calculated by:

$$T_0 \leftarrow \frac{-\Delta U_g^*}{\ln x_0} \tag{4}$$

where ΔU_g^* is a significant difference in global utility, and x_0 is the desired probability of visiting a significantly inferior neighbor at the start of the search. The exponential cooling parameter, α, can be computed by:

$$\alpha \leftarrow \left(\frac{-\Delta U_g^*}{T_0 \ln x_{b-1}}\right)^{\frac{1}{b-1}} \tag{5}$$

where b is the number of evaluations in the search budget and x_{b-1} is the desired final probability of visiting a significantly inferior neighbor.

5.5 Evolutionary Programming

Evolutionary programming employs the paradigm of evolution to evolve improved solutions. The algorithm uses a parent population (size M) to generate an offspring population. The first step in this algorithm is to generate an initial population. In the architecture search, the initial population is comprised of mutated copies of the starting architecture.

In the service selection search, the initial population is randomly generated. Then evolutionary programming enters a loop of the following steps:

1. Select the M solutions with the highest fitness (predicted U_g).
2. Move surviving solutions to parent population.
3. Parent solutions reproduce to generate offspring population of size K.
4. Mutate offspring solutions.
5. Determine fitness (predicted U_g) of offspring solutions.

This loop continues until the search budget is consumed. If the populations are *overlapping*, offspring and parent solutions compete for survival in step 1. If the populations are *non-overlapping*, only offspring are eligible for being selected in step 1. Reproduction in evolutionary programming is asexual and an offspring is initially an identical copy of the parent.

Evolutionary programming uses a phenotypic representation, so the features of the solution are mutated directly. The size of the mutation is influenced by a parameter called the *step size*. When a solution mutates, the number of changes made to the architecture is randomly generated from a normal distribution $N(\mu, \sigma)$ with μ set to the step size and σ set to 0.5μ (a minimum of one change per mutation is enforced). The type of change made to a software architecture $\mathcal{A}$ is randomly selected from the following list:

- A change in the level of a security option.
- A change to the architectural pattern of one component.
- Increasing by one the number of service instances in a composite component.
- Decreasing by one the number of service instances in a composite component.

The changes made in service selection mutation are substitutions of SPs. We use an adaptive step size in service selection search. This means that the step size itself is modified by adding a randomly selected value from a normal distribution with μ set to zero and σ set to a parameter called the *adaptive step factor*. Employing adaptive step size allows the search to make large jumps through the space at the start of the search. When a near-optima is located, individuals with more modest mutations will tend to have the highest fitness, and consequently individuals with smaller step sizes are likely to be favored. As the step sizes shrink, the search converges on the near-optima and moves from exploration to exploitation.

5.6 Challenges of Heuristic Search

Each heuristic search algorithm has its own strengths and weaknesses. The global utility landscape of the architecture and service selection solution spaces can vary in ruggedness (i.e., the number of local optima and the shapes of these optima). This ruggedness is difficult to quantify and may change with each re-architecting event. Each heuristic algorithm will strongly interact with the ruggedness of the utility landscape in a different way. The parameter selection (e.g., filter settings, population settings, step sizes) will have its own interactions with the landscape.

In a SASSY system with no meta-controller, a system administrator would need to select a heuristic algorithm for both the architecture search and the service selection search. The system administrator would have three options:

1. use an educated guess to select heuristic algorithms,
2. tinker with heuristic settings until adequate performance is achieved, or
3. run detailed time-consuming tests to find optimal heuristic settings.

Guessing runs the risk of making a poor choice in heuristic algorithms that would consequently lead to poor performance in re-architecting. The second and third options are labor intensive and require a skilled system administrator, which is antithetical to the goals of autonomic computing. Therefore, we propose automating the selection of heuristic algorithms and their parameters with a meta-controller.

6. META-CONTROLLER

The primary function of the meta-controller, the second major contribution of this paper, is to decide the heuristic algorithms and their parameters at the start of a re-architecting event. The meta-controller has two auxillary functions to support its decision-making process: 1) training on previously encountered problems and 2) analyzing the collected performance data from the training process. The meta-controller contains a *candidate list* of pre-existing heuristic search combinations (one for architecture selection and the other for service selection) that were found to be successful in other SASSY applications. Ideally, the candidate list should contain a variety of search algorithms.

When the meta-controller makes a heuristic selection decision in a re-architecting event, the meta-controller stores an optimization problem, $\mathcal{P}$, consisting of the starting architecture and a list of all the SPs with their current QoS metrics. After the re-architecting search completes, the meta-controller stores a *result tuple* of $\mathcal{P}$: $(\mathcal{H}_{arch}, \mathcal{H}_{SrvSel}, U_g^{\text{best}})$. After the re-architecting process completes, the meta-controller begins a preemptive training process testing other heuristic search combinations from the candidate list against $\mathcal{P}$ and storing the outcome in the result tuple.

Below we present two different designs for the meta-controller, each with its own analytic method and decision-making process.

6.1 Overall Best Heuristic Pair

The `Overall Best` meta-controller attempts to determine the overall best candidate heuristic combination over the entire range of re-architecting optimization problems encountered by a SASSY application. Each time a result tuple is stored, the `Overall Best` meta-controller updates the average U_g^{best} for the given heuristic combination. When it is time for the `Overall Best` meta-controller to make a decision, it chooses the heuristic combination that has produced the highest average U_g^{best}.

6.2 Context Best Heuristic Pair

It is unlikely that there is a single heuristic search combination that outperforms all other heuristic search combinations over the potential optimization problem space. A certain heuristic combination may dominate a portion of the

optimization problem space, while other heuristic combinations dominate other portions of the space. The `Context Best` meta-controller attempts to determine the overall best candidate heuristic combination given specific features of $\mathcal{P}$.

In many architecture search problems, a near-optimal architecture may be nearby the starting architecture. When facing such problems, the autonomic controller is best served by using heuristic search algorithms that intensely scan the architecture space surrounding the starting point. In other architecture search problems, the closest near-optima are relatively far away from the starting architecture. With these problems, the autonomic controller is better served using heuristic search algorithms that can travel some distance from the starting architecture.

Changes in the service environment that have occurred since the previous re-architecting can impact the expected distance of near-optimal architectures from the starting architecture. Thus, measurements of service environment changes may offer insight into the likelihood of proximate near-optimal architectures. These metrics can be used as features in a machine learning problem. If the meta-controller can successfully train on these features via a machine learning approach, the meta-controller may be able to predict whether an exploitative (e.g., beam search) or exploratory heuristic search algorithm (e.g., simulated annealing) is more likely to be successful.

Changes in QoS metrics and utility scores may be useful features in predicting whether the architecture search and service selection searches should employ neighborhood filtering for the local search algorithms. It is possible that machine learning approaches may make other connections between optimization problem features and heuristic combinations.

6.2.1 Characterizing the Optimization Problem

An accurate and relevant representation of the optimization problem is required for a machine learning approach to successfully train. The representation used for the `Context Best` meta-controller presented in this paper is shown in Table 1. The features in the *Component* group and *Security Option* group reflect the starting architecture of the system and some statistics on the service environment. The BSC architectural pattern stands for a basic component in the architecture, while the LB architectural pattern represents a load-balancing composite component in the architecture, and the fFT architectural pattern indicates a fast fault-tolerant composite component [26]; one and only one of these three fields must be set to true for each component. The current level field in the *Security Option* group is set to the level of security enabled on a component for that particular security option (multiple security options may be specified by the domain expert). The *Overall*, *SSS utility*, and *QoS Metric* groups reflect the performance of the architecture and service selection in the current service environment.

6.2.2 Processing the Training Set

Whenever a result tuple is stored, the `Context Best` meta-controller extracts the features of the problem, $\mathcal{F}(\mathcal{P})$, in Table 1. A training set record, keyed to $\mathcal{F}(\mathcal{P})$, is created that contains an empty linked-list of result tuples. The training set record is then added to the training set's specialized data structure (this data structure has both hash table and array properties). If the training set already contains a matching training record, the new results are appended to the pre-existing record in the training set.

6.2.3 Decision Making

When the `Context Best` meta-controller needs to select a candidate heuristic combination, it extracts $\mathcal{F}(\mathcal{P}_{current})$ for the current re-architecting problem. If a training record with a matching $\mathcal{F}(\mathcal{P})$ is found in the training set, the `Context Best` meta-controller determines which candidate heuristic combination has the best recorded performance in that training record.

If no such training record exists, the `Context Best` meta-controller employs the k-nearest neighbor (KNN) algorithm [13] as follows:

1. Calculate the Euclidean distance between $\mathcal{F}(\mathcal{P}_{current})$ and the key of each training record.
2. Select top k closest training records.
3. Each of the k training records votes for the candidate heuristic that performed best on its problems.
4. If one heuristic combination received more votes than any other, select that heuristic combination. If there is a tie in the voting, select the heuristic combination from the training record closest to $\mathcal{F}(\mathcal{P}_{current})$.

7. EXPERIMENTAL EVALUATION

This section describes experiments used to assess the two different meta-controllers: `Overall Best` and `Context Best`. A third meta-controller that randomly selects a heuristic pair was used as a control.

7.1 Problem Application and Environment

Each meta-controller was assigned to manage an SOA application of 25 components. Each component is considered to have its own service type. For each component, we randomly generated about six possible SPs. No component had fewer than three possible SPs. Fig. 3 shows the data-flow diagram of the managed application. Each meta-controller was given a cost constraint that afforded roughly 2.25 SPs per component. Each SP has the following attributes:

- capacity (i.e., the maximum transaction rate for the provider),
- execution time,
- availability, and
- cost.

No SP was dominated in all four attributes by another provider.

7.2 SSSes and Utility Functions

The SSSes and utility functions were randomly generated for the application depicted in Fig. 3. The random generation process ensures that no two SSSes share both the same QoS metric and the same pathway through the application. Table 2 shows the SSSes used in the experimental evaluation.

Each security option has three levels. The lowest level has no impact on an SP's capacity or execution time. Increases in security levels reduce an SP's capacity and lengthen the execution time.

Group	Value	Type	Number of Features
Overall	U_g	floating point	1
Overall	$\Delta(U_g)$	floating point	1
SSS Utility	$U(q)$	floating point	n_{SSS}
SSS Utility	$\Delta(U(q))$	floating point	n_{SSS}
Component	BSC Arch. Pattern	boolean	n_{cmp}
Component	LB Arch. Pattern	boolean	n_{cmp}
Component	fFT Arch. Pattern	boolean	n_{cmp}
Component	number of SPs used	integer	n_{cmp}
Component	number of SPs available	integer	n_{cmp}
Component	number of SPs changed	integer	n_{cmp}
QoS Metric	current q for component	floating point	$n_{cmp} \times n_{QoS}$
QoS Metric	$\Delta(q)$ for component	floating point	$n_{cmp} \times n_{QoS}$
Security Option	current level	integer	$n_{cmp} \times n_{sec}$

Table 1: Features of the machine learning problem.

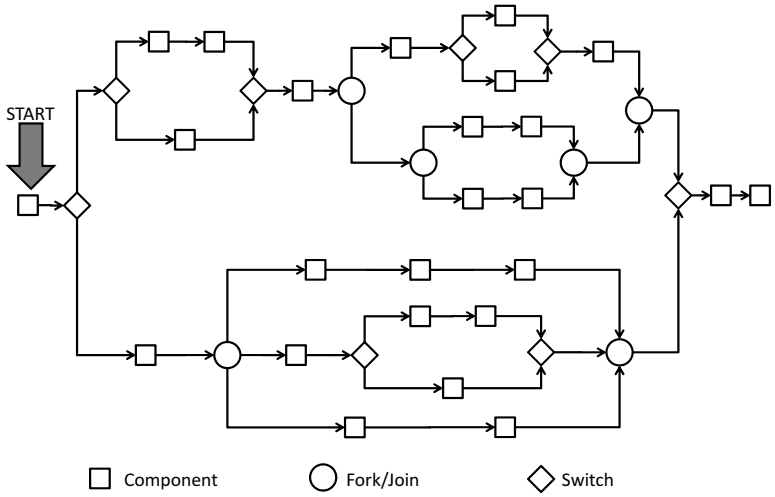

Figure 3: SOA application for experimental evaluation.

QoS Metric	Weight	Number of Components
Security Option 1	0.13	14
Security Option 2	0.06	13
Security Option 2	0.06	13
Security Option 2	0.07	14
Security Option 2	0.10	17
Throughput	0.11	16
Throughput	0.17	13
Throughput	0.05	13
Availability	0.17	16
Execution Time	0.08	14

Table 2: SSSes used in experimental evaluation.

Each security option SSS considers its current security level to be the lowest level found on any member component. To determine their utility, security option SSSes use a discrete utility payoff table.

The throughput, availability, and execution time SSSes use sigmoidal utility functions similar in form to those found in [26].

7.3 Candidate Heuristic Combinations and Other Meta-Controller Settings

During the development process, the heuristic search algorithms and meta-controller procedures were tested and debugged with a 30-component SOA application not presented here. Using the development SOA application, a metaheuristic genetic algorithm was employed to find the following four near-optimal heuristic search algorithms for the architecture search:

1. an opportunistic hill-climber (HC) with SSS filter, $k = 5$, and component filter, $j = 2$,
2. beam search (BS) with beam width of two, SSS filter, $k = 5$, and component filter, $j = 2$,
3. evolutionary programming (EP) with non-overlapping populations, parent population size $M = 6$, offspring population size $K = 30$, and a step size of 2.0, and
4. simulated annealing (SA) with $p(V_{init}^{inf})$ set to 66% and $p(V_{last}^{inf})$ to 0.0023% (V^{inf} is defined here as a move with a -0.1 change in U_g.)

A similar metaheuristic genetic algorithm, also not presented here, was employed to find two near-optimal heuristic search parameters for the service selection search:

1. an opportunistic hill-climber (HC) with no neighborhood filtering and
2. evolutionary programming (EP) with overlapping populations, parent population size $M = 3$, offspring population size $K = 19$, initial step size of 3.5, and an adaptive step factor of 4.5.

The four architecture heuristic search algorithms were combined with the two service selection heuristic algorithms to make eight heuristic search combinations: 1) `HC-HC`, 2) `HC-EP`, 3) `BS-HC`, 4) `BS-EP`, 5) `EP-HC`, 6) `EP-EP`, 7) `SA-HC`, and 8) `SA-EP`.

Each of the architecture searches was configured to run with 5 threads, and each of the service selection searches was configured to run with 25 threads. Composite components were limited to a maximum size of five basic components. The architecture search budget was set to 100 architecture evaluations. The service selection search budget was set to 1,200 service selection evaluations. With these budget settings, a re-architecting search should take less than 1 minute on most modern computers.

In both `Overall Best` and `Context Best`, the number of training replications for each problem encountered was set to one. The `Context Best` KNN algorithm was run with $k = 5$. The re-architecting threshold was set to 80% of U_g predicted during the last re-architecting.

To provide a baseline for comparison, we built a third meta-controller, `Random`, that randomly selects one of the eight heuristic search combinations described above to employ whenever a re-architecting event occurs. We also tested simple autonomic controllers (i.e., no meta-controller used) that always use the same heuristic search combination.

7.4 Simulation

Each simulation commenced with the SOA application in a near-optimal architecture determined a priori by an 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 MTTF (Mean Time To Failure). 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 MTTR (Mean Time To Repair) 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 meta-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 each meta-controller.

7.5 Experimental Results

The meta-controllers encountered about 230 re-architecting events on average over the course of a single simulation run. We calculated the average U_g over the course of each simulation experiment. Figure 4 shows the distribution of average global utilities in each set of 100 experiments produced by the eight simple controllers and three meta-controllers. The boxes in this figure show the three population quartiles, while the whiskers show the maximum and minimum. Next we assess the statistical significance of the results.

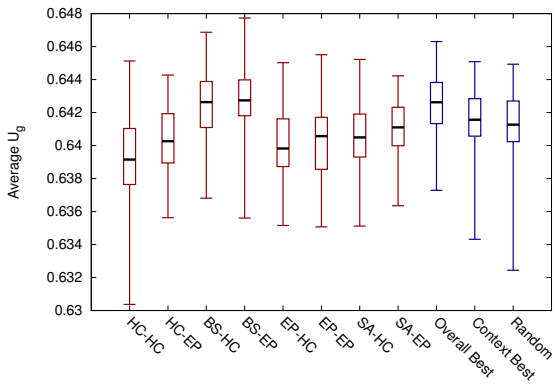

Figure 4: Box plot showing the quartiles of the simulation runs.

Table 3 shows the mean of the average global utility for each of the meta-controllers along with 95% confidence intervals. The meta-controller `Overall Best` clearly outperforms the other two meta-controllers; its lower confidence bound is greater than the upper bounds of the others.

The small range of average U_g is due to the meta-controllers keeping the systems at near-optimal U_g most of the time. Occasionally, a critical SP will fail, and it is either not possible or very difficult to achieve near-optimal U_g. The overall duration of failure events causing more than a 10% reduction in U_g was observed to be less than 15 ϵ in the average simulation run. Though uncommon, the differences in meta-controller response to these failures result in some statistical differences in the average U_g.

Controller	Lower Bound	Mean	Upper Bound
`HC-HC`	0.63858	0.63910	0.63962
`HC-EP`	0.63994	0.64035	0.64076
`BS-HC`	0.64194	0.64234	0.64273
`BS-EP`	0.64225	0.64268	0.64311
`EP-HC`	0.63960	0.64001	0.64043
`EP-EP`	0.63987	0.64028	0.64069
`SA-HC`	0.64014	0.64054	0.64095
`SA-EP`	0.64062	0.64098	0.64134
`Ovrll Bst`	0.64228	0.64263	0.64297
`Cntxt Bst`	0.64129	0.64164	0.64200
`Random`	0.64081	0.64119	0.64157

Table 3: 95% confidence intervals for net overall average U_g.

We applied the Tukey-Kramer procedure to perform a simultaneous pair-wise comparison of the eleven controllers (eight simple controllers and three meta-controllers) in Table 3. We determined that `Overall Best` was significantly better than `Random` and six of the eight simple controllers (those employing `HC-HC`, `HC-EP`, `EP-HC`, `EP-EP`, `SA-HC`, and `SA-EP`) at the 95% confidence level. This procedure also demonstrated that `Context Best` was better than five of the

simple controllers (HC-HC, HC-EP, EP-HC, EP-EP, and SA-HC). We further applied the Tukey-Kramer procedure on just the results of the three meta-controllers, and we were able to conclude that Overall Best was significantly better than Context Best and that Context Best was significantly better than Random at the 95% confidence level.

Figure 5 shows the U_g over time. All three meta-controllers do well in maintaining U_g over the course of the simulation runs. As can be seen by the size of the error bars, the experimental variance makes it difficult to compare the different meta-controllers; this variance cancels out to some degree when computing the overall average for each simulation experiment.

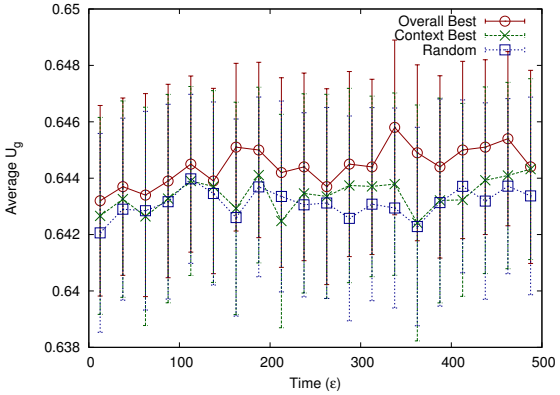

Figure 5: The average global utility over time with 95% error bars.

Table 4 shows the heuristic combination performance data collected by the Overall Best meta-controller. Heuristic combinations employing hill-climbing for service selection search perform poorly in comparison to heuristic combinations employing evolutionary programming for service selection search. The best heuristic combination is BS-EP, using beam search in the architecture search and evolutionary programming in the service selection search.

Heuristic Search Combination	Lower Bound	Mean	Upper Bound
HC-HC	0.6305	0.6310	0.6314
HC-EP	0.6412	0.6415	0.6419
BS-HC	0.6367	0.6370	0.6374
BS-EP	0.6429	0.6432	0.6435
EP-HC	0.6402	0.6405	0.6408
EP-EP	0.6414	0.6417	0.6420
SA-HC	0.6377	0.6381	0.6385
SA-EP	0.6414	0.6417	0.6420

Table 4: Average heuristic combination peformance tables collected by Overall Best with 95% confidence intervals.

The evolving behavior of the meta-controllers can be seen in Fig. 6. The data series labeled *early* in Fig. 6 were collected from just the first half of the simulation, while the data series labeled *late* were collected from only the second half of the simulation. We can see that by the second half of the simulation, Overall Best has clearly converged on to the most overall effective heuristic combination, BS-EP.

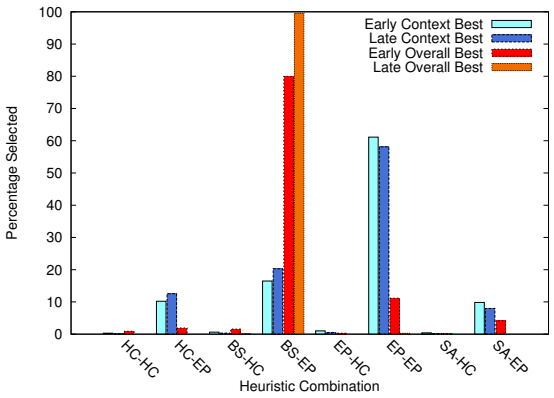

Figure 6: Percentage of time a heuristic combination was selected by the meta-controllers.

To better understand the differences in heuristic combination selection between Context Best and Overall Best, we collected 2,000 re-architecting problems encountered in our simulations. Each heuristic combination was tested against each problem 30 times. For each problem, the average U_g found by all the heuristic combinations was calculated. Then, the relative performance of each heuristic combination on each problem was determined.

Figure 7 shows a scatter plot of the relative performance of BS-EP vs EP-EP on each re-architecting problem. The thin black line shown in Fig. 7 indicates where the performance of BS-EP and EP-EP are equal; a large concentration of problems are close to this line. EP-EP outperformed BS-EP on 78.2% of the problems. However, as can be seen in Fig. 7, when BS-EP outperforms EP-EP, it is typically by a larger margin. The average difference in U_g between EP-EP and BS-EP when EP-EP is better equals 0.00048 whereas when BS-EP is better the difference is 0.01268, approximately 25 times greater.

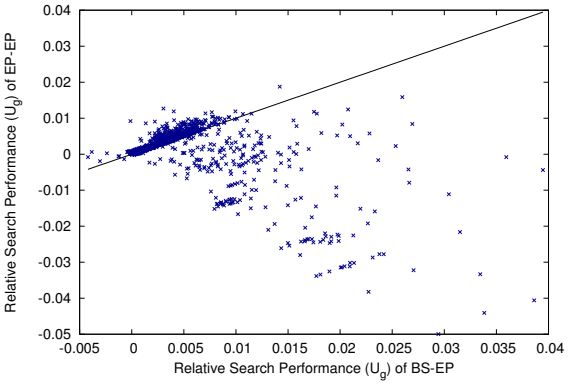

Figure 7: Scatter plot of relative heuristic combination performance on 1,935 re-architecting problems.

The KNN algorithm used in the Context Best meta-controller does not account for the risk that picking incorrectly EP-EP over BS-EP could lead to a relatively large drop in performance. The Context Best meta-controller would need to correctly identify the best heuristic combination about 85% of the time to equal the performance of Overall Best. It is unlikely that simple KNN can achieve such accuracy in the face of the following challenges presented by the use of online training sets:

- a relatively small number of training problems,
- one training replication per heuristic combination for each problem, and
- a relatively large number of fields in the problem characterization.

The experiments were performed on systems with two 2.4 GHz quad-core hyper-threading Intel Xeon processors. Re-architecting searches using hill-climbing for service selection took an average of 15.0 seconds to complete, while re-architecting searches using evolutionary programming for service selection took an average of 6.9 seconds to complete. The choice of architecture heuristic algorithm had less impact on re-architecting search times.

8. RELATED WORK

In [2], 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 some SPs to be white boxes, and it can modify the configuration parameters and resource allocations for those white box SPs. QosMOS does not consider architectural patterns for improving QoS. Optimization in QoSMOS is conducted through exhaustive search, a technique that would not scale well to the problems considered by SASSY.

Cardellini et al. devise a framework, MOSES, for optimizing SOA systems in [3]. 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. The use of LP limits the form of the objective function in MOSES. SASSY does not face similar restrictions on the form of the utility function. On larger problems, MOSES must restrict the space of substitutions considered to keep the problem solvable in near real-time.

Mani et al. in [21] develop a system using Role Based Modeling Language to model the performance impact of design pattern changes in SOA systems. As the SOA application implements a new design pattern, the changes in the systems are passed to the system's performance model.

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 [19], employs architectural tactics in a multi-objective evolutionary algorithm to expedite the multi-objective search process. Martens et al. present a similar system in [22] that starts quickly by using LP on a simplified version of the problem to prepare a starting population for a multi-objective evolutionary algorithm.

9. CONCLUSION

The `Overall Best` meta-controller showed a statistically significant benefit over the other two meta-controllers. Although `Overall Best` did not outperform simple controllers using `BS-HC` and `BS-EP`, it was not known a priori which heuristic search combinations would provide the best performance. The `Overall Best` meta-controller was able to identify `BS-EP` as the best heuristic search combination without having to run its own large batch of simulation experiments. The cost of training the `Overall Best` meta-controller is very small. Therefore, using this meta-controller for non-trivial SOA software systems would provide a net measurable benefit, improving the performance of the system and reducing the burden on human administrators.

The relatively poor performance of the `Context Best` meta-controller was initially surprising. Further analysis revealed that the `Context Best` meta-controller was unable to adjust for the risk presented by disparities in relative heuristic combination performance.

If the `Context Best` meta-controller could identify appropriate heuristic search combination with a high level of accuracy while accounting for risk, `Context Best` would likely provide superior performance. The `Context Best` meta-controller may be improved by upgrading the machine-learning technique employed from simple KNN to a more advanced method. Machete and similar methods provide advanced versions of the KNN algorithm [13]. Another possibility would be to substitute either a neural network or an SVM. These have the disadvantage of requiring time to train, but SVM allows the use of penalty weights to control the risk of selecting the wrong heuristic search combination.

In future work, we plan to test our approaches on a wider variety of SOA applications. We also plan to model the lifetime utility U_l, which would reflect the utility produced by an architecture over its expected lifetime. Modeling U_l and incorporating adaptation costs would provide a more holistic approach to assessing architectures and service selections.

Acknowledgements

The work of D. Menascé is partially supported by NIST grant No. 70NANB12H277.

10. REFERENCES

[1] M. N. Bennani and D. A. Menascé. Assessing the robustness of self-managing computer systems under highly variable workloads. In *Proc. 1st IEEE International Conference on Autonomic Computing (ICAC '04)*, pages 62–69, New York, NY, May 2004.

[2] 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*, 37(3):387–409, 2011.

[3] 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*, 38(5):1138–1159, 2012.

[4] E. Casalicchio, D. A. Menascé, V. Dubey, and L. Silvestri. Optimal service selection heuristics in service oriented architectures. In *Quality of Service in Heterogeneous Networks*, pages 785–798. Springer, 2009.

[5] B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. *Software engineering for*

self-adaptive systems: A research roadmap. Springer, 2009.

[6] A. D'Ambrogio. Model-driven WSDL extension for describing the QoS of web services. In *IEEE International Conference on Web Services (ICWS '06)*, pages 789–796, Chicago, IL, Sept. 2006.

[7] V. Dubey and D. A. Menascé. Utility-based optimal service selection for business processes in service oriented architectures. In *IEEE International Conference on Web Services*, pages 542–550, Miami, FL, July 2010.

[8] A. Elkhodary. A learning-based approach for engineering feature-oriented self-adaptive software systems. In *Proc.* 18^{th} *ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 345–348, 2010.

[9] A. Elkhodary, N. Esfahani, and S. Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *Proc.* 18^{th} *ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 7–16, 2010.

[10] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, Upper Saddle River, NJ, 2005.

[11] N. Esfahani, S. Malek, D. A. Menascé, J. P. Sousa, and H. Gomaa. A modeling language for activity-oriented composition of service-oriented software systems. In *Proc. 12th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems MODELS '09*, pages 591–605, Denver, CO, Oct. 2009.

[12] J. M. Ewing and D. A. Menascé. Business-oriented autonomic load balancing for multi-tiered web sites. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '09)*, pages 279–288, London, United Kingdom, Sept. 2009.

[13] J. Friedman. Flexible metric nearest neighbor classification. Technical report, Stanford University Statistics Department, 1994.

[14] 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*, pages 462–469, Sierre, Switzerland, Mar. 2010.

[15] M. C. Huebscher and J. A. McCann. A survey of autonomic computing–degrees, models, and applications. *ACM Computing Surveys*, 40(3):1–28, Aug. 2008.

[16] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlicting, and C. Pu. Generating adaptation policies for multi-tier server applications in consolidated server environments. In *Proc. 5th IEEE International Conference on Autonomic Computing (ICAC '08)*, pages 23–32, Chicago, IL, June 2008.

[17] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, Jan. 2003.

[18] T. G. Kolda, R. M. Lewis, and V. Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review*, 45(3):385–482, Sept. 2003.

[19] A. Koziolek, H. Koziolek, and R. Reussner. Peropteryx: automated application of tactics in multi-objective software architecture optimization. In *QoSA-ISARCS '11*, pages 33–42, 2011.

[20] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering (FOSE '07)*, pages 259–268, Minneapolis, MN, May 2007.

[21] N. Mani, D. C. Petriu, and M. Woodside. Propagation of incremental changes to performance model due to soa design pattern application. In *Proc. ACM/SPEC international conference on International conference on performance engineering*, pages 89–100. ACM, 2013.

[22] 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 G. Heineman, J. Kofron, and F. Plasil, editors, *Research into Practice–Reality and Gaps*, volume 6093 of *LNCS*, pages 84–101. Springer Berlin Heidelberg, 2010.

[23] D. A. Menascé, E. Casalicchio, and V. Dubey. A heuristic approach to optimal service selection in service oriented architectures. In *Proc. 7th International Workshop on Software and Performance (WOSP 2008)*, pages 13–24, Princeton, NJ, June 2008.

[24] D. A. Menascé, E. Casalicchio, and V. Dubey. On optimal service selection in service oriented architectures. *Performance Evaluation Journal*, 67(8):659–675, Sept. 2009.

[25] D. A. Menascé, R. Dodge, and D. Barbará. Preserving QoS of e-commerce sites through self-tuning: A performance model approach. In *Proc. 3rd ACM Conference on E-commerce*, pages 224–234, Tampa, FL, Oct. 2001.

[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 *Workshop on Software and Performance*, pages 27–36, San Jose, CA, Jan. 2010.

[27] D. A. Menascé, H. Gomaa, S. Malek, and J. Sousa. Sassy: A framework for self-architecting service-oriented systems. *IEEE Software*, 28(6):78–85, Nov. 2011.

[28] 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)*, pages 195–204, Washington, DC, June 2010.

[29] N. Poggi, T. Moreno, J. L. Berral, R. Gavaldá, and J. Torres. Self-adaptive utility-based web session management. *The International Journal of Computer and Telecommunications Networking*, 53(10):1712–1721, July 2009.

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

[31] L. Zhang and D. Ardagna. SLA based profit optimization in autonomic computing systems. In *Proc. 2nd International Conference on Service Oriented Computing (ICSOCŠ04)*, pages 173–182, New York, NY, Nov. 2004.