AUTONOMIC PERFORMANCE OPTIMIZATION WITH APPLICATION TO SELF-ARCHITECTING SOFTWARE SYSTEMS

by

John M. Ewing

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

_____ Dr. Daniel Menascé, Dissertation Director

_____ Dr. Hassan Gomaa, Committee Member

_____ Dr. Sam Malek, Committee Member

_____ Dr. Stephen G. Nash, Committee Member

_____ Dr. Sanjeev Setia, Department Chair

_____ Dr. Kenneth S. Ball, Dean
Volgenau School of Engineering

Date: _____ Spring Semester 2015
George Mason University
Fairfax, VA

Autonomic Performance Optimization with Application to Self-Architecting Software Systems

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

John M. Ewing
Master of Science
Illinois Institute of Technology, 2003
Bachelor of Science
University of Richmond, 1997

Director: Dr. Daniel Menascé, Professor
Department of Computer Science

Spring Semester 2015
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to my wife, Theresa.

Acknowledgments

I would like to thank first and foremost my advisor, Daniel Menascé. His guidance, support, and suggestions have proven invaluable.

I'd also like to thank the other members of the SASSY team, Hassan Gomaa, João Sousa, Sam Malek, Naeem Esfahani, Koji Hashimoto, Zeynep Zengin, and Minseong Kim. Their ideas, concepts, and work provided the foundation and the context for this dissertation.

I would like to express my gratitude to Hassan Gomaa, Sam Malek, and Stephen Nash for serving on my dissertation committee.

I would also like to thank the National Science Foundation for supporting me as a graduate research student under the SASSY project (grant CCF-0820060) and George Mason University's Provost Office for their support during part of my doctoral work.

Carlotta Domeniconi and Sean Luke shared valuable insights on machine learning and heuristic search algorithms in my conversations with them.

I'd also like to thank Vasudeva Akula for sharing his experimental data during my work on autonomic load-balancing.

My mother-in-law, Russetta Canavan, was tremendously supportive during her many visits—I could not have finished without her help.

My father, Lionel Ewing, spent many hours collecting debugging information to help me identify a rare race-condition fault in the multi-threaded re-architecting search.

Over the years, my sister, Margaret Ewing, and my sister-in-law, Dolores Prin, offered much encouragement.

The final push to finish this dissertation would not have been possible without the babysitting assistance of Helen Maher and Noli Candelaria. They altered their schedules many times over the last few months to accommodate my deadlines.

My friend, Harold White, lent a listening ear as I hunted bugs and made various hypotheses. I'd also like to thank him for his considerable support during the week of my final defense.

I'd like to acknowledge my children, Alexander, Daniel, and Sarah, who have lived with "Daddy's dissertation" for all of their lives. Watching Alexander learn to program, teaching Daniel about plots, and singing dissertation ditties to Sarah brought me joy during this process.

Finally, a most special thank you to my wife, Theresa, for her support, forbearance, and editing skills.

Table of Contents

List of Tables

List of Figures

Abstract

AUTONOMIC PERFORMANCE OPTIMIZATION WITH APPLICATION TO SELF-ARCHITECTING SOFTWARE SYSTEMS

John M. Ewing, PhD

George Mason University, 2015

Dissertation Director: Dr. Daniel Menascé

Service Oriented Architectures (SOA) are an emerging software engineering discipline that builds software systems and applications by connecting and integrating well-defined, distributed, reusable software service instances. SOA can speed development time and reduce costs by encouraging reuse, but this new service paradigm presents significant challenges. Many SOA applications are dependent upon service instances maintained by vendors and/or separate organizations. Applications and composed services using disparate providers typically demonstrate limited autonomy with contemporary SOA approaches. Availability may also suffer with the proliferation of possible points of failure–restoration of functionality often depends upon intervention by human administrators.

Autonomic computing is a set of technologies that enables self-management of computer systems. When applied to SOA systems, autonomic computing can provide automatic detection of faults and take restorative action. Additionally, autonomic computing techniques possess optimization capabilities that can leverage the features of SOA (e.g., loose coupling) to enable peak performance in the SOA system's operation. This dissertation demonstrates that autonomic computing techniques can help SOA systems maintain high levels of usefulness and usability.

This dissertation presents a centralized autonomic controller framework to manage SOA systems in dynamic service environments. The centralized autonomic controller framework

can be enhanced through a second meta-optimization framework that automates the selection of optimization algorithms used in the autonomic controller. A third framework for autonomic meta-controllers can study, learn, adjust, and improve the optimization procedures of the autonomic controller at run-time. Within this framework, two different types of meta-controllers were developed. The `Overall Best` meta-controller tracks overall performance of different optimization procedures. `Context Best` meta-controllers attempt to determine the best optimization procedure for the current optimization problem. Three separate `Context Best` meta-controllers were implemented using different machine learning techniques: 1) K-Nearest Neighbor (`KNN MC`), 2) Support Vector Machines (SVM) trained offline (`Offline SVM`), and 3) SVM trained online (`Online SVM`).

A detailed set of experiments demonstrated the effectiveness and scalability of the approaches. Autonomic controllers of SOA systems successfully maintained performance on systems with 15, 25, 40, and 65 components. The `Overall Best` meta-controller successfully identified the best optimization technique and provided excellent performance at all levels of scale. Among the `Context Best` meta-controllers, the `Online SVM` meta-controller was tested on the 40 component system and performed better than the `Overall Best` meta-controller at a 95% confidence level. Evidence indicates that the `Online SVM` was successfully learning which optimization procedures were best applied to encountered optimization problems. The `KNN MC` and `Offline SVM` were less successful. The `KNN MC` struggled because the KNN algorithm does not account for the asymmetric cost of prediction errors. The `Offline SVM` was unable to predict the correct optimization procedure with sufficient accuracy—this was likely due to the challenge of building a relevant offline training set. The meta-optimization framework, which was tested on the 65 component system, successfully improved the optimization techniques used by the autonomic controller.

The meta-optimization and meta-controller frameworks described in this dissertation have broad applicability in autonomic computing and related fields. This dissertation also details a technique for measuring the overlap of two populations of points, establishes an approach for using penalty weights to address one-sided overfitting by SVM on asymmetric data sets, and develops a set of high performance data structure and heuristic search templates for C++.

Chapter 1: Introduction

This chapter provides an introduction to the technologies and challenges that are addressed in this dissertation.

1.1 The Challenge of Service-Oriented Architectures

This section discusses Service-Oriented Architectures (SOA): flexible, emerging technologies that solve many problems but also introduce new management challenges.

1.1.1 The Need for New Computing Paradigms

Advances in computer and network technology over the last thirty years have helped to produce an information revolution. Today, the recording, processing, and consumption of digital information pervades most aspects of daily life and has led to substantial improvements in personal and organizational productivity. As users have become accustomed to and dependent upon information technology (IT), they have developed higher expectations of computer system functionality, performance, and availability. As a result, providers of information technology are under pressure to rapidly develop and deploy new functionality at reduced cost. To cope with such demands, IT managers are embracing new paradigms such as SOA.

1.1.2 The SOA Approach

Whereas the Object-Oriented Architecture (OOA) paradigm views application design from a data perspective, the SOA paradigm views application design from a functional perspective. When distributed computing technologies like Web Services are combined with SOAs, previously developed service modules and already deployed service providers can be easily reused and incorporated into new computing applications [100]. This ability to reuse existing services encourages collaborative development efforts across multiple organizations and

is enabling service marketplaces to emerge where vendors offer the use of a managed service provider for a fee. Reuse of existing services reduces costs and time required to develop and deploy SOA applications–in fact, true SOA systems can be composed from services discovered at runtime [10].

The key principles to enabling the benefits of service-orientation are [35]:

- *Loose Coupling*–Service modules are designed such that dependencies between services are minimized.
- *Service Contract*–Service providers create and maintain relationships through the establishment of service contracts.
- *Autonomy*–Service providers control how they render service and are robust to the failures of other service providers.
- *Abstraction*–The logic of service modules and the platforms of service providers are hidden from the outside world.
- *Reusability*–The size and contents of service modules should encourage reuse.
- *Composability*–The design of an SOA should allow new applications to be created from previously existing service providers.
- *Statelessness*–Services avoid maintaining state information.
- *Discoverability*–Service providers provide discoverable descriptions of the service.

1.1.3 The Challenge of Autonomy in SOA

A significant challenge for IT managers is that many SOA applications will be dependent upon service instances maintained by vendors and/or separate organizations. Using contemporary SOA approaches, applications and composed services using disparate providers typically demonstrate limited autonomy. Poor performance or failure of a single service provider frequently impact the overall SOA applications dependent upon that provider. Thus, to meet the high performance expectations of users, IT staff must vigilantly monitor all service providers used by their SOA applications. If a service provider suffers from

performance degradation or failure, IT staff must respond quickly to replace the service provider with another functionally equivalent service provider. In some service degradation and failure cases, IT staff may be required to rearchitect some SOA applications to meet performance objectives–a process that could take human administrators minutes to hours. Thus, the cost of staff effort and the complexity of monitoring and maintaining SOA technologies become major considerations in decisions regarding the fielding of SOA applications. In SOA environments, where the performance and availability of service providers are highly dynamic, human management of the application architecture and service selection is likely infeasible. Autonomic computing could provide a powerful remedy to the problem of maintaining SOA applications by automating the monitoring, repair, and even optimization of SOA applications [102].

1.2 Overview of Autonomic Computing

A longstanding tradition in the worlds of computer science and artificial intelligence (AI) is to draw inspiration from nature to find effective solutions to difficult problems. This approach has yielded many great discoveries, including neural networks, simulated annealing, and the field of evolutionary computation. This decade has seen a new movement, autonomic computing, which is inspired by the autonomic functions of living organisms. Complex organisms depend on subconscious management of many critical, ongoing biological processes (e.g., adjusting the organism's heart rate) while their conscious minds can focus on higher level tasks (e.g., finding and catching food). Applying this paradigm to computer technology, computer systems manage their own essential low-level functions, thereby freeing human users, developers, and administators to focus on higher-level tasks [64].

The field of autonomic computing emerged in the late 1990's and early 2000's as researchers realized well-defined and routine tasks, such as parameter tuning, could be handled by computers themselves. This promising field of research has quickly gained momentum and now represents a major initiative involving many disciplines, including AI, systems engineering, software engineering, operations research, and Information Technology (IT) management.

1.2.1 Self-* Properties

Autonomic computing seeks to provide information systems with the following capabilities [42, 64]:

1. *A reduction in the Total Cost of Ownership (TCO).* Autonomic computing aims to reduce the TCO by decreasing the human effort required to manage complex systems and by more efficiently using both hardware and energy resources [53, 71].

2. *Improvements in Quality of Service (QoS) metrics.* Measurements that assess user experience with the system, such as response time and availability, comprise QoS metrics. By making near-optimal decisions, autonomic systems can improve QoS metrics thereby enhancing the user experience [89].

3. *Ability to adapt in the face of environmental change.* Autonomic systems should detect significant and possibly unexpected changes in their environment and respond appropriately. This adaptation may include modification of the autonomic system's behavior and even the system's structure [22, 69].

4. *Ability to evolve in the face of new requirements and functionality.* The introduction of new requirements by users and the addition of new features by developers should be seamlessly integrated into the autonomic system with minimal disruption and effort [69].

5. *Enhanced ability to defend against attacks.* Autonomic systems can enhance security through automated response to detected attacks and through the development of dynamic trust models for external entities [27, 56].

To achieve the five capabilities detailed above, researchers have proposed four central self-* properties that autonomic systems should exhibit: self-configuration, self-optimization, self-healing, and self-protection [64]. To be considered self-configuring, an autonomic system must be able to receive high-level operational goals and then successfully move to a configuration satisfying these goals. Some configurations satisfy goals better than others, and self-optimizing autonomic systems will find those configurations that best satisfy

operational goals in both the present and the future. Systems with the self-healing property should diagnose both internal and external problems as they occur and then implement appropriate remedies. Self-protecting autonomic systems monitor for attacks and take steps to defend themselves in a proactive manner [56].

1.2.2 Models of Autonomic Function

Autonomic systems can be considered intelligent agents [56]. Russell and Norvig [114] propose an intelligent agent model in which intelligent agents collect a sequence of percepts from their environments via sensors. The agents apply functions to the percept sequence to generate a set of actions that are effected via actuators [114]. Similar models like the Sense-Plan-Act (SPA) model have been used in the field of robotics [69].

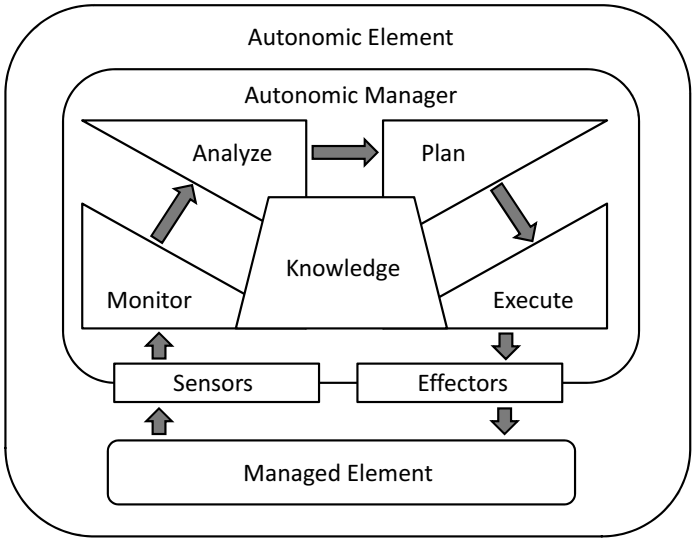


Figure 1.1: Depiction of the IBM reference model for autonomic computing, MAPE-K.

In a further refinement of the intelligent agent model, IBM introduced the Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) model. In this model, autonomic systems are comprised of autonomic elements. Each autonomic element contains an autonomic

manager and a managed element (see Fig. 1.1). As in the intelligent agent model, the autonomic manager receives percepts from the managed element through sensors. The monitoring process filters these percepts and passes relevant data to the analysis process. The analysis process transforms the raw data to information usable in a decision-making process. If the result of the data analysis warrants action, the usable information is passed to the planning process. The planning process finds an improved configuration of the system and develops a sequence of actions that will move the managed element to the desired configuration. Finally, the effector implements the sequence of actions ordered by the planning process [56, 57].

A cornerstone of the MAPE-K model is the knowledge module. Each of the processing steps within the autonomic manager interacts with the manager's knowledge module. In the case of monitoring, knowledge may provide insight on percept filtering. For analysis, knowledge may provide insight on which estimation algorithm should be used. For planning, knowledge is essential for predicting the impact of an action. Knowledge can consist of both pre-existing information and learned information [56, 57].

Inspired by recent advances in robotic agent models, Kramer and Magee [69] suggest a more flexible autonomic model for managing architectures. Rather than impose a processing loop like SPA or MAPE-K, they suggest a three layer model consisting of the component control layer, the change management layer, and the goal management layer. The interaction between the layers is message-driven. At the lowest level, the component control layer interacts directly with the managed element, collecting and filtering percepts, and effecting simple changes requested by the upper layers. The change management layer is able to make rapid decisions via pre-compiled plans reminiscent of event-condition-action tuples [43, 56]. The plans are implemented either in response to monitoring data from the component control layer or in response to a request by the goal management layer. The lower layers enable the goal management layer to perform searching of the configuration space and perform high-level planning to ascertain the sequence of actions needed to achieve a near-optimal configuration. A significant advantage of this model is that it allows the autonomic manager to operate simultaneously on different time scales. For example, a system may have percepts arrive on a millisecond time scale, but making and implementing a plan

may take tens of seconds.

1.2.3 Degrees of Autonomicity

Huebscher and McCann [56] 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 of 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.

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.

System and software architectures can be configured, optimized, healed, and protected by autonomic managers. Recent work suggests that viewing the architecture as an autonomically managed element may provide the best method for enabling all four self-* properties in the autonomic system [56, 69].

1.2.4 Utility Functions

Utility functions are frequently used by intelligent agents to represent preferences and/or high-level objectives [114]. In autonomic computing, utility functions were introduced in [125] as a way of representing business objectives in autonomic computing systems. Generally, utility functions associate scalar values representing usefulness with system states.

For many systems, intelligent agents need to express preferences regarding several distinct attributes. In many autonomically managed systems [7, 37, 63, 92, 125], a separate utility function is specified for each attribute of interest. For an autonomically managed SOA application, such attributes might consist of security features and QoS metrics such as execution time, availability, and throughput [92].

By combining the results of attribute utility functions, a global utility function can produce a single utility value indicating the overall usefulness of a system state [7, 37, 92]. When using global utility functions, autonomic agents seek to maximize the global utility over time.

1.3 Problem Statement

There is a need for methods and mechanisms that provide autonomic capabilities to SOA applications operating in environments that exhibit unpredictable workload intensities, service and communication failures, and have QoS requirements specified by utility functions. These autonomic capabilities should reduce the human effort of managing SOA applications.

1.4 Thesis Statement

The thesis statement is broken into the three related statements listed below.

H1: Autonomic computing techniques can be used to automatically design the architecture of SOA-based software systems and select service providers in a scalable way that optimizes utility.

H2: Scalable autonomic computing techniques that self-adapt to new architectures and new service provider selections can be used to maintain optimized levels of utility for SOA-based software systems in the face of failures and performance degradations of service providers.

H3: An autonomic meta-controller employing machine learning techniques can measure and tune the autonomic controller's performance, and thus reduce the human effort required to manage the system.

1.5 Success Criteria

The following success criteria will be used to evaluate the hypotheses.

C1: Develop techniques for the automatic design of architectures and corresponding service selection and demonstrate through rigorous experimentation that the methods achieve optimized solutions in a scalable way.

C2: Develop techniques for the automatic re-architecting and corresponding service selection for SOA-based systems in the face of failures and performance degradation. Demonstrate through rigorous experimentation that the methods maintain high utility solutions in a scalable way.

C3: Develop a meta-heuristic agent that adaptively selects heuristics and their parameters to solve self-architecting and service selection optimization problems.

1.6 Contributions

This dissertation makes the following major contributions:

1. *A centralized autonomic framework* that supports goal management, change management, and performance monitoring for optimizing architectures in SOA applications.

2. A general approach for discovering *near-optimal goal architectures* through predictions of utility.

3. Goal management that takes steps toward *full autonomicity* through adaptive selection of heuristic search and search parameters.

4. *An experimental evaluation* of the developed autonomic frameworks.

This dissertation also makes the following minor contributions:

1. A general approach for the meta-optimization of autonomic systems.

2. A general method for measuring the overlap of two populations of points.

3. A reasoned approach for using penalty weights to address one-sided overfitting by support vector machines (SVM) on asymmetric data sets.

4. High performance C++ templates that combine the best features of arrays, linked lists, and hash tables.

5. A library of high performance C++ heuristic search templates.

Chapter 2: Background

This chapter provides necessary background information about the concepts and technologies that will be utilized by this dissertation. The first section discusses various approaches to autonomic computing. Performance evaluation of computer systems with a focus on SOA systems is addressed in the second section. The third section considers different approaches to optimization and examines heuristic search algorithms. Machine learning techniques are described in the fourth section. The fifth section provides an overiew of Self-Architecting Software SYstems (SASSY). Finally, the sixth section surveys related work.

2.1 Approaches to Autonomic Computing

This section provides background on three major approaches to autonomic computing: control theory, artificial intelligence (AI) techniques, and heuristic search supported by modeling.

2.1.1 Control Theory

Control theory is a mature engineering discipline for maintaining some set of reference properties in a dynamic system through the use of feedback loops [25]. Control theoretic principles help machinery maintain stability in unstable environments. Many autonomic systems based on control theory have been developed [25, 26, 50, 53, 71, 85, 103, 104]. Control theory works well when there is a control parameter in the system that can be continously adjusted. For example, in web servers, control theory might continuously adjust the `MAXTHREADS` parameter to keep CPU utilization below some threshold [25].

Criticisms of control theory in autonomic computing can be found in [52] and [56]. The argument presented in [52] is that there are substantial risks of negative interactions between control theoretic autonomic components in medium to large systems–these negative interactions can be subtle and difficult to foresee. In [56], it is pointed out that control

theoretic approaches are tightly coupled to implementation–this makes autonomic systems employing control theory rigid and difficult to evolve. In situations where no continuous control parameter is available (e.g., architecture management), control theory is reduced to managing systems through simplistic event-condition-action rules where events are generated when a reference variable crosses some threshold. Another limitation of control theory is that it is intended for maintaining system stability and not system optimization.

2.1.2 AI Technologies

Many of the technologies developed in AI can be adapted to autonomic computing. The more commonly applied techniques are discussed in this section.

Bayesian networks, a form of machine learning, provide mechanisms for computing complex interactions of conditional probabilities [114]. Using bayesian networks, empirical observations of the system and environment are analyzed and converted into knowledge. Autonomic agents use this knowledge to support the planning and decision-making. Knowledge obtained through bayesian networks has been applied to enhancing QoS in SOA [74,126,133]

Fuzzy logic allows reasoning with gradations of true and false [114]. Fuzzy logic has machine learning applications and can be used to learn relationships between parameters and metrics in autonomic systems [130]. Knowledge developed via fuzzy logic has been used to support the decision-making of autonomic agents managing SOA systems in [75, 98].

Reinforcement learning observes the consequences of actions taken by an autonomic agent [124]. These observations are used to infer the relative utility of the actions taken. Reinforcement learning has been applied to autonomic management of SOA systems in [61]. Reinforcement learning can spend considerable time exploring and learning the consequences of various actions. Autonomic controllers employing reinforcement learning may take considerable time to develop optimized behaviors.

Supervised learning uses labeled training sets to support decision making in autonomic systems [98]. Supervised learning will be considered in more detail in Section 2.4.

A knowledge base (KB) in AI is a repository of facts and deductions. Reasoning enabled by a KB is sometimes used to drive planning and decision-making in autonomic systems [106].

2.1.3 Heuristic Search Supported by Modeling

Many autonomic controllers are based on search that is supported by performance models [56]. The performance models provide the capability to predict the performance of any potential system configuration. Then, a search algorithm is employed to explore the system configuration space in a quest for the most suitable system configuration. Often, the most suitable system configuration is the configuration 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 those situations where the assumptions of the performance model do not hold [6].

Parameter tuning was an early application of adaptive systems employing heuristic search [89]. Heuristic search has proven a popular choice for resource allocation [7, 37, 60, 107, 132]. Recent work with these methods has focused on selecting at run-time optimal software architectures [92] and service selections [16, 30, 87, 88].

2.2 Performance Evaluation

This section defines key Quality of Services (QoS) metrics used to assess SOA application performance, describes performance models that accurately predict QoS values in SOA applications, and discusses utility functions for assessing the usefulness of different QoS performance levels.

2.2.1 QoS Metrics

The following three QoS metrics represent the majority of users' concerns about performance in SOA computer applications:

- execution time (t_e),
- availability (a), and
- throughput (x).

Similar to response time from the client/server paradigm, execution time is the amount of time that passes while a service is being rendered. The execution time can be derived

by comparing the time stamp for the receipt of the service request message, T_r, and the time stamp for the sending of a service response message, T_s. It should be noted that in SOA systems, a single service request message can generate multiple service response messages. Thus, in SOA systems, an execution time is always associated with a specific service response message. This is reflected in equation 2.1 through the use of subscript i.

$$t_{e,i} = T_{s,i} - T_r. \tag{2.1}$$

Availability is the fraction of time that an application or service is up (i.e., able to process service requests in a normal manner). Availability can be computed from uptime, t_u, and downtime, t_d, as follows:

$$a = \frac{t_u}{t_u + t_d}. \tag{2.2}$$

If the mean time to failure, $\bar{t}_f$, and the mean time to repair, $\bar{t}_r$, are known for a service provider, the expected availability of the provider, $\mathrm{E}[a]$, can be calculated as shown in equation 2.3.

$$\mathrm{E}[a] = \frac{\bar{t}_f}{\bar{t}_f + \bar{t}_r}. \tag{2.3}$$

Throughput is the rate at which service requests are successfully processed. Throughput can be computed from the number of successfully completed requests, N_{req}^{cmp}, over period of time, t, as shown in equation 2.4.

$$x = \frac{N_{req}^{cmp}}{t}. \tag{2.4}$$

2.2.2 Performance Modeling of SOA Systems

Performance models predict how different system factors affect QoS metrics [86]. Within autonomic computing systems, analytic performance models are used to predict QoS values for different system configurations [6, 7, 37, 89, 92]. Modeling of SOA systems reflects the principles of loose coupling, autonomy, and abstraction by dividing performance models into two categories: 1) service producer models and 2) service consumer models.

Service Producer Models

Service producers are the service providers that receive and process service requests. Given inputs of service demands, arrival rates, and available resources, service producer models predict execution times and/or availabilities of service providers. These models are used by the administrators of a service provider to maintain the service provider's performance. Service producer models are also used to determine the advertised QoS parameters in standards such as Web Service Level Agreement (WSLA) [76].

Service Consumer Models

Service consumer models consider service instances as black boxes and do not speculate on the internal dynamics of service instances. Instead, service consumer models focus on predicting availability, throughput, and execution time of process flows given the advertised availabilities, capacities, and execution times of the service instances comprising the application flow.

Process flows in SOA applications can be represented in tree data structures [95]. The leaf nodes of the tree are invocations of service instances. Process flow constructs such as sequential flows and fork-and-joins serve as parent nodes in the tree structures [95]. Process flow tree structures can be converted to expression trees that act as service consumer performance models. This tree conversion process substitutes operators for parent nodes and QoS metric distributions for leaf nodes.

2.2.3 Utility Functions

As mentioned in chapter 1, utility functions provide a means for intelligent agents to determine the usefulness of system states. Within autonomic computing, utility functions are most commonly applied in the data-driven and heuristic search approaches but can also be employed by control theoretic approaches. This subsection discusses attribute utility functions and global utility functions in more detail.

Attribute Utility Functions

Attribute utility functions represent either the utility of enabling a certain feature or the utility of achieving a particular level of QoS. There is typically a discrete utility bonus associated with each feature type (e.g. security). Table 2.1 shows a sample utility function for an encryption feature that supports Data Encryption Standard (DES), Triple DES, and Advanced Encryption Standard (AES).

Encryption Type	Security Level	Utility
unencrypted	none	0.1
DES	low	0.3
Triple DES	medium	0.8
AES	high	1.0

Table 2.1: Sample discrete utility function for encryption levels.

In order to optimize data rates for different network flows, network researchers frequently apply utility functions to QoS metrics [1,72,120]. A typical concave utility function applied to data rates is the logarithmic utility shown in equation 2.5:

$$U_i(v) = \gamma + \alpha \log(v + \beta) \tag{2.5}$$

where v is the measured QoS metric, α is a scaling constant, and β and γ are constants used to move the origin of the logarithmic function. This utility function can be adapted for the throughput QoS metric in some SOA applications (see Fig. 2.1).

Concave utility functions for network flow data rates yield convex network optimization problems amenable to mathematical programming techniques [120]. Thus, most work on network optimization has used concave utility functions for flow data rate. However, concave utility functions are only appropriate for QoS metrics that are elastic [1,72]. A QoS metric is considered elastic within a computer application if that application always experiences decreasing marginal utility degradation as the value of the QoS metric worsens [116], as

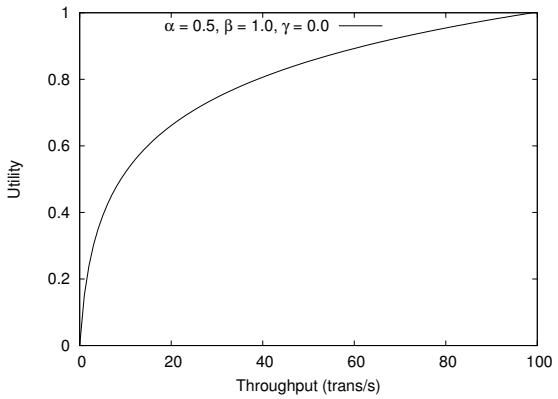


Figure 2.1: An example of a logarithmic utility function for throughput.

shown in Fig. 2.1. As modern users have become sensitive to degradation of execution time and availability, these metrics are considered to be inelastic for interactive SOA applications. For many interactive SOA applications, throughput is sometimes considered to be inelastic. For inelastic QoS metrics, a sigmoid utility function is often appropriate [1,7,63,92,114,116, 125]. Sigmoid utility functions are convex in regions where the metric value is poor (i.e., not meeting some QoS goal) and concave in regions where the metric value is good (i.e., exceeding some QoS goal). Thus the inflection point of the sigmoid utility function occurs at the QoS goal. A sigmoid utility function for execution time, throughput, and availability is shown in equation 2.6 [92].

$$U_i(v) = K_i \frac{e^{\alpha_i\ (\beta_i - v)}}{1 + e^{\alpha_i\ (\beta_i - v)}} \tag{2.6}$$

where K_i is a normalizing factor equal to

$$K_i = \begin{cases} (1 + e^{\alpha_i \beta_i})/e^{\alpha_i \beta_i} & \text{for execution time} \\ 1 & \text{for throughput} \\ (1 + e^{\alpha_i (\beta_i - 1)})/e^{\alpha_i (\beta_i - 1)} & \text{for availability,} \end{cases} \tag{2.7}$$

β_i is the QoS goal for the metric v and α_i is a sensitivity parameter that defines the sharpness of the curve. The sign of α_i determines whether the sigmoid decreases ($\alpha_i > 0$) with v or increases ($\alpha_i < 0$) with v. The maximum value of this sigmoid utility function is 1. These maximum values occur at $v = 0$ for execution time, $v = 1$ for availability, and for $v \to \infty$ for throughput. Typically, a decreasing utility function is used for execution time and an increasing one is used for throughput and availability (see Fig. 2.2). Automated procedures for determining appropriate QoS goals in sigmoid utility functions are available [91].

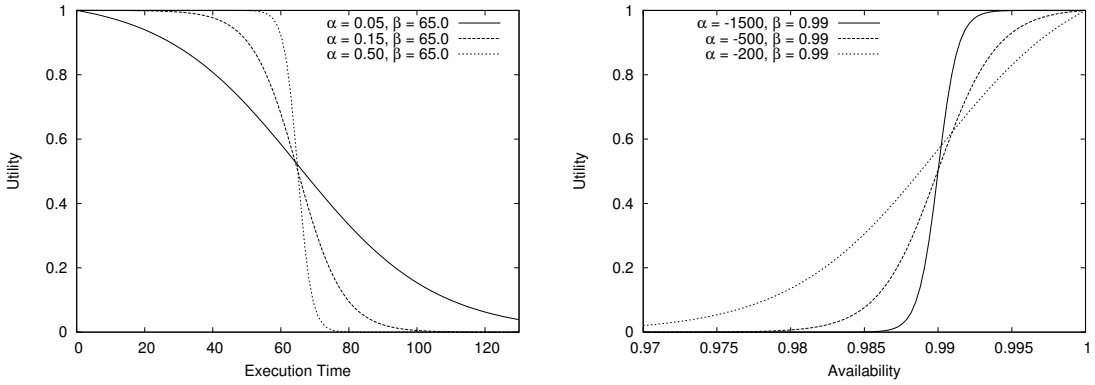


Figure 2.2: The effect of α on execution time and availability sigmoid utility functions.

Global Utility Functions

A global utility function provides a mechanism for combining separate attribute utilities into a single utility score. There are myriad possibilites for defining global utility functions; this section will discuss two common, practical approaches–a weighted arithmetic mean and a weighted geometric mean. The weighted arithmetic mean is straightforward:

$$U_g = \frac{\sum_{i=1}^{N} U_i \times w_i}{\sum_{i=1}^{N} w_i} \tag{2.8}$$

where U_g is the global utility value, N is the number of attributes, U_i is the utility for attribute i, and w_i is the weight for attribute i. Using an arithmetic mean as a global utility function can lead to solution topologies where the optimal solution maximizes some attribute utilities while neglecting the utilities of other attributes. Such solutions may not be desirable for many SOA applications where failure to meet any QoS objective may make a substantial negative impression on the user. For example, if the optimal solution maximizes throughput and minimizes execution time but neglects availability, users are likely to consider the application unusable.

In contrast, a weighted geometric mean is likely to create solution topologies where the optimal solution attempts to maximize each attribute utility according to its weight [92].

$$U_g = \left(\prod_{i=1}^{N} U_i^{w_i} \right)^{1/\sum_{i=1}^{N} w_i} \tag{2.9}$$

It is also possible to use both the arithmetic and geometric means to create various hybrid utility functions where some utilites are added and others are multiplied. *Preference logic*, an emerging system evaluation subdiscipline, studies the development of such hybrid utility functions [31,32]. Preference logic rationalizes global utility functions through nesting and blending of disjunctions, arithmetic means, and geometric means. The key characteristic of an attribute in preference logic is its *andness* (which is inversely related to the attribute's *orness*). If an attribute utility score is essential to overall system performance, then that attribute is considered to have a high level of andness (and consequently a low level of orness). If an attribute utility score matters only when other attribute utility scores are low, then that attribute is considered to have a high level of orness (and consequently a low level of andness) [31]. An arithmetic mean is used to combine attributes that have equal levels of andness and orness, while a geometric mean is used to combine attributes with high andness and low orness [32].

Finally, it should be noted that the selection of attribute and global utility functions has substantial impact on the characteristics of the utility topology in the solution space. Some combinations of utility functions will create relatively smooth utility landscapes over the

solution space, while other combinations of utility functions will produce challenging, rugged utility landscapes. The topology of the expected utility landscapes must be considered in the selection of optimization algorithms.

2.3 Optimization and Heuristic Search

Optimization of configurations and resource allocations is a fundamental goal in many engineering and management disciplines. The purpose of optimization is to find values for a vector of decision variables, $\vec{x}$, that maximize (or in some problems minimize) an objective function $f(\vec{x})$ subject to a set of contraints, C. By observing the properties of $f(\vec{x})$ and C, optimization problems can be categorized into the following three types:

- linear,
- convex non-linear, and
- non-convex non-linear.

Maximizing global utility for an SOA application is typically a non-convex non-linear optimization problem.

2.3.1 Linear Programming Optimization

When $f(\vec{x})$ is a linear function to be optimized, and C is a set of linear constraints, the problem can be categorized as a linear optimization problem [122]. Linear problems can be optimized through a set of techniques known as linear programming. A linear optimization problem can be represented in either of two different forms: a *primal* problem that is typically a minimization problem (e.g. cost minimization) and a *dual* problem that is typically a maximization problem (e.g. profit maximization) [122]. The objective function to be minimized in the primal problem is recast as a set of constraints in the dual problem, and the constraints of the primal problem are recast as an objective function to be maximized in the dual problem [122]. The dual problem is sometimes easier to solve and can provide insight into the solution of the primal problem.

In both primal and dual formulations, linear programming techniques require non-negative values for each decision variable, x_i, comprising $\vec{x}$ [48]. The feasible points of the solution space are the values of $\vec{x}$ that comply with all of the constraints in C–these feasible points are bounded by a polytope (i.e., an n-dimensional polygon) defined by the linear constraints [122]. The linear objective function $f(\vec{x})$ can be thought of as defining a continuum of parallel hyperplanes [96, 122].

In linear programming minimization problems, the smallest value of $f(\vec{x})$ that results in an objective function hyperplane touching the constraint polytope is the minimum value of $f(\vec{x})$ [96, 122]. The solution to the minimization problem is the point of intersection between this objective function hyperplane and the constraint polytope, $\vec{x}^*$ [96, 122]. In linear programming maximization problems, the objective function hyperplane is defined by the largest value of $f(\vec{x})$ that touches the constraint polytope.

In both minimization and maximization problems, the intersection is guaranteed to occur at a corner of the polytope (in some cases it may be possible for the intersection to include an edge or a face of the polytope) [96,122]. The number of corners on the polytope is combinatorial, $\binom{n}{m}$, where n is the number of decision variables and m is the number of linear constraint equations [122]. A common method for solving linear programming problems is the simplex method, which travels from corner to corner of the polytope. The simplex algorithm starts at one corner of the polytope and then assesses the gradient offered by each edge of the polytope connected to its current corner [96, 122]. The simplex algorithm follows the edge offering the steepest descent in minimization problems or the steepest ascent in maximization problems [96, 122]. The simplex algorithm follows this edge until it reaches the next corner, where it again evaluates the edges connected to its position. This process continues until the simplex algorithm reaches a corner where no edge offers further improvement; this corner is returned by the simplex algorithm as the minimum or maximum depending on the goal [96, 122]. The worst case time complexity for the simplex algorithm is exponential with regard to n [48, 96]. However, the average time complexity of the simplex algorithm is polynomial with regard to n [48, 96].

2.3.2 Non-Linear Optimization

Convex non-linear optimization problems have convex constraint sets and seek to either minimize a non-linear convex objective function or maximize a non-linear concave objective function. Such problems are guaranteed to have a single optimum, and a number of techniques including Newton's method, quasi-Newton methods, and extensions of the simplex method have been developed for solving such problems in polynomial time [48, 96, 114].

Non-convex non-linear optimization problems contain at least one of the following:

- a non-convex constraint set,
- a non-convex objective function to be minimized, or
- a non-concave objective function to be maximized.

Non-convex non-linear optimization problems are inherently difficult to optimize because there may exist multiple local optima [112]; typically, only one of these optima is the true global optimum or supremum [96]. The methods used on convex problems, like the simplex algorithm, are likely to converge to the nearest optimum regardless of whether it is a mere local optimum or the global optimum.

A global utility function expressed by a combination of functions similar to those found in equations 2.6, 2.8, and 2.9 is unlikely to be concave. Thus, the problem of maximizing global utility for an SOA application will typically belong to the class of non-convex non-linear optimization problems. Exhaustive search of the configuration space faced by an autonomic controller of an SOA application is typically NP-hard (see Chapter 4) and consequently impractical for supporting online decisions. More pragmatic search methods are required if an autonomic controller searching configuration spaces is to succeed.

2.3.3 Heuristic Search Overview

NP-hard optimization problems are often solved through heuristic search techniques [7,114]. There are three major categories of heuristic search techniques: A*, local search, and evolutionary algorithms [114]. This project utilizes local search and evolutionary algorithms. A* search depends upon having an admissible cost heuristic function (an admissible cost

heuristic function estimates the eventual total cost of a partial solution/configuration without ever overestimating the cost) [114]. In optimization problems, this cost includes the difference in utility between the current partial solution and the optimal solution. Since the utility of the optimal solution is seldom known a priori, it is rare to find admissible cost heuristic functions for optimization problems. Most research on NP-hard optimization problems has focused on local search algorithms and evolutionary algorithms [110]. It should be noted that local search algorithms and evolutionary algorithms are not guaranteed to find the global optimum, however in most cases these heuristic algorithms do find near-optimal solutions. Sacrificing an optimal solution for a near-optimal solution is usually an acceptable tradeoff to avoid 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 global utility with respect to either the number of evaluations or processing time consumed by the search.

For the autonomic controller presented in Chapter 4 the utility landscape 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.

The next section outlines heuristic search algorithms that may be applied in autonomic computing. Many of these have tunable parameters that can be used to modify the behavior

and performance of the heuristic.

2.3.4 Local Search

Local search algorithms (known as direct search in the operations research community [65]) 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.

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. Some work has considered pruning the neighborhood through the use of heuristic filtering (see Section 4.5.1). A neighborhood heuristic filter examines the shortcomings of the currently visited solution and identifies and visits only those neighbors who are most likely to have an improved global utility score.

Hill-Climbing

Hill-climbing is a simple local search heuristic algorithm 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 to find the neighboring solution with the highest utility score before visiting any neighboring solutions. If the greedy hill-climber finds a neighboring solution with a higher score than the currently visited solution, the greedy hill-climber will visit the neighboring solution with the highest evaluation score and then generate a new neighborhood. An opportunistic hill-climber evaluates each member 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 neighboring 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.

Beam Search

Beam search is a more sophisticated local search heuristic algorithm that concurrently visits multiple solutions. 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*, k. 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. One of the following distinct criteria may be applied before finalizing the k candidate solutions as the next level list:

1. Candidate solutions must exceed the highest score amongst solutions on the previous level list.
2. Candidate solutions must equal or exceed the lowest score amongst solutions on the previous level list.
3. All candidate solutions are accepted as the next level list.

Using criterion 1 will lead beam search to ascend optima in a more exploitative manner, but with this criterion beam search may become trapped in local optima (i.e., an empty level list) more frequently. Under criterion 2, beam search may more slowly ascend optima by more fully exploring the solution space, but this extra exploration may help beam search avoid some local optima. It should be noted that using criterion 2, beam search will need to store some of the evaluated solutions in memory to avoid visiting the same solutions over and over while looping through the solution space. With criterion 3, beam search will never become trapped in local optima, though on some landscapes the extra exploration may prove costly. Also, beam search using criterion 3 will need to store all evaluated solutions in memory to avoid endless loops through the solution space. As in hill-climbing, when beam search becomes trapped in a local optimum, it may randomly generate a new level list and

recommence the search. The implementation of beam search used in Chapters 4, 5, and 6 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.

Simulated Annealing

Simulated annealing is a stochastic local search heuristic algorithm inspired by the cooling techniques used to strengthen solid materials such as steel and glass. A physical annealing process reduces temperatures according to a schedule that helps ensure the crystal lattices of the material will settle into low energy states, thus reducing the fragility of the lattices. The simulated annealing search heuristic operates like opportunistic hill-climbing with one key difference: simulated annealing may stochastically decide to visit inferior (i.e., lower predicted U_g) neighbors [110]. This stochastic behavior is meant to simulate the jittering of molecules as they settle into crystal lattices. In simulated annealing, the probability of visiting an inferior neighbor i is determined as follows [110]:

$$p\left(V_i^{inf}\right) = e^{\left(\frac{-\Delta U_g^i}{T}\right)} \tag{2.10}$$

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. An exponential cooling schedule is commonly employed:

$$T_{i+1} \leftarrow \alpha T_i \tag{2.11}$$

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:

$$T_0 \leftarrow \frac{-\Delta U_g^*}{\ln x_0} \tag{2.12}$$

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:

$$\alpha \leftarrow \left(\frac{-\Delta U_g^*}{T_0 \ln x_{b-1}} \right)^{\frac{1}{b-1}} \tag{2.13}$$

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.

2.3.5 Evolutionary Algorithms

Local search heuristic algorithms such as hill-climbing–although highly effective on certain topologies–can suffer poor performance on more rugged topologies because their intense focus on a small portion of the solution space fails to consider the bigger picture of the entire solution space. Since only small variations separate a visited solution from its neighboring solutions, neighboring predicted utilities are expected to be correlated. Thus, local search algorithms can suffer poor performance because they only examine one neighborhood at a time [110], which can lead to wasted processing time when excessive effort is spent performing evaluations in poor neighborhoods. Additionally, the local search algorithms hill-climbing and beam search are easily trapped in local optima [114].

Evolutionary algorithms use paradigms from nature to develop a broader view of the solution space [24, 62, 96, 110]. Traditional evolutionary algorithms such as evolutionary programming, evolution strategies, and genetic algorithms base their search approach on the idea of natural selection. More recently developed evolutionary algorithms such as particle swarm and ant colony optimization are inspired by patterns of communication

found in the cooperative behavior of organisms.

Within the traditional evolutionary algorithms, solutions are referred to as individuals and the properties of the individual are referred to as the phenotype. Some evolutionary algorithms use an encoded representation of these properties called the genotype. Either the phenotype or the genotype must be used to represent the solution in an evolutionary algorithm. The predicted U_g of a solution is termed the individual's fitness, and changes to individuals are called mutations. Most evolutionary algorithms start with a randomly selected population of individuals. These algorithms then assess the fitness of each individual in the starting population. The traditional algorithms next enter a birth/death cycle of creating new individuals and removing individuals.

The first step in this cycle is to select the parents based upon some set of rules. For example, an algorithm could be directed to select only the fittest 25% of individuals (defined by U_g) in the population to be parents. The second step is for the parents to produce offspring. An offspring is typically either a copy of a parent or a blend of two or more parents, modified by some stochastic mutation operation. The algorithms use another set of rules to determine how many offspring each parent produces (this is sometimes referred to as the brood size). After all of the offspring are evaluated, the algorithms have a procedure to determine which individuals from the previous population and which offspring will survive to form the population of the next generation. If offspring compete with parents to form the next generation, the population is said to be overlapping [24]. The birth/death cycle begins anew with the selection of parents from this new generation. Subsequent generations are created until either a specified threshold fitness has been achieved or a specified budget of evaluations has been completely consumed.

Evolutionary Programming

Originally developed for the optimization of finite state machines, evolutionary programming techniques have been applied to a wide range of optimization problems [96]. Evolutionary programming heuristic search algorithms typically use a phenotypic representation of solutions, so the features of the solution are able to be mutated directly. The search is initialized with randomly generated individuals forming an initial population of size M [24].

In each iteration of the birth/death cycle in evolutionary programming, every member of the population produces a single offspring, which is initially an identical copy of the parent. The properties of the individual are directly mutated. The probability of mutations introducing small changes is high, while the probability of mutations introducing large changes is low. In phenotypic evolutionary algorithms, the *average step size* is the average difference produced by a mutation [24]. Typically, evolutionary programming uses overlapping populations, where the offspring and parent populations are combined together before competing for survival. Next, evolutionary programming applies a survival selection method called *truncation* where the population is sorted by fitness, and the top M individuals are retained as the parent population in the next iteration [24]. This process continues until a stopping criterion is satisfied (either threshold fitness achieved or evaluation budget consumed).

Evolution Strategies

Similar in many respects to evolutionary programming, evolution strategies was originally developed for optimization of spray nozzle designs [24]. Typically, evolution strategies begin with a small population (size M) of randomly generated individuals. In each iteration of the birth/death cycle in evolution strategies, a relatively large offspring population of size K (where $K >> M$) is generated by having each member of the previous population produce a number of offspring equal to the brood size of $\frac{K}{M}$ (typically the broodsize is an integer). Evolutionary programming can be thought of as a special case of evolution strategies where the populations are overlapping and $M = K$ [24]. Evolution strategies uses a phenotypic representation of individuals, and the evolutionary programming approach to mutation is also adopted here. In evolution strategies, the parent population may be discarded or used as part of an overlapping population. Truncation is again used for survival selection. As in evolutionary programming, this process continues until a stopping criterion is satisfied (either threshold fitness achieved or evaluation budget consumed).

Adaptive step size is an advanced feature often employed in evolution strategies [5, 24]. With adaptive step size enabled, each individual in the population maintains an average step size parameter for each dimension of the solution space. Along with the phenotypic

representation, these average step size parameters are passed on to offspring. When using adaptive step sizes, mutation becomes a two-step process. First, each of the individual's average step size parameters is stochastically modified. Second, the individual's phenotypic representation undergoes mutation in accordance with its new step sizes. On some utility landscapes, the use of adaptive step size can substantially accelerate the algorithm's convergence.

Evolution strategies will sometimes employ a technique called *recombination* during reproduction [5]. When using recombination, a specific method adapted to the solution space (e.g. arithmetic mean) converts the M solutions of the parent population into a single representative solution. This representative solution then produces K offspring solutions which are then mutated.

Genetic Algorithms

Seeking to more closely emulate the process of natural selection in living organisms, genetic algorithms manipulate genotypic representations of individuals [24]. An individual's genotype is a string or set of strings that encode the properties of the individual's phenotype. Binary strings are a frequent form of representation in genotypes. As with other evolutionary algorithms, genetic algorithms begin with a randomly generated initial population of size M. In each iteration of the birth/death cycle an offspring population of size M is produced. Typically in genetic algorithms, offspring solutions are produced from a blending of two parent solutions. Parents are selected for each offspring by one of three stochastic methods: 1) fitness proportional selection, 2) linear rank selection, or 3) tournament selection.

In fitness proportional selection, each time the genetic algorithm needs to select a parent, a weighted *roulette wheel* is spun. In fitness proportional selection, the probability of the roulette wheel stopping on (and thereby selecting) a population member i is [24, 110]:

$$P(i) = \frac{U_g^i}{\sum_{i=1}^{M} U_g^i} \tag{2.14}$$

When employing fitness proportional selection, a genetic algorithm's rate of convergence is correlated to the breadth of the population's fitness distribution. At the start of a genetic algorithm search, the randomly generated population's fitness distribution is frequently widely dispersed. Thus early in the search, fitness proportional selection will tend to cause the genetic algorithm to rapidly converge to better solutions. However, as the population converges toward a concave maximum (as is frequently the case for global utility optima), the fitness distribution of the population will become tightly clustered. Hence, final convergence to a concave utility maximum is often a leisurely process when using fitness proportional selection [24].

In linear ranking, the population is first sorted in descending order according to fitness (U_g). As in fitness proportional selection, a roulette wheel is also employed. The probability of the roulette wheel stopping on sorted population member i is [24, 110]:

$$P(i) = \frac{1+\mathcal{S}}{M} - \frac{2\mathcal{S}(i-1)}{M(M-1)} \tag{2.15}$$

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. Typically with moderate to large values of $\mathcal{S}$, linear ranking selection will slow the early convergence of the genetic algorithm, while speeding up the final convergnce on concave maxima–these are considered desirable features on many utility landscapes. A drawback of linear ranking selection is that the computational cost of sorting large populations can slow down the search.

Tournament selection provides some of the convergence rate benefits of linear ranking selection at a reduced computational cost [24]. In tournament selection, each time the genetic algorithm needs to select a parent, a tournament is formed by choosing q participant solutions from the population with uniform probability [24]. The winner of a tournament is the solution with the highest fitness; this winning solution is then selected as a parent.

Regardless of whether fitness proportional, linear ranking, or tournament selection is used, the selected parent solutions are then blended to produce offspring using a process

called *crossover* in which the strings comprising the genotypes of the parents are spliced together [24, 110]. Three different types of crossover are commonly employed in genetic algorithms: 1-point, 2-point, and uniform [24]. In 1-point crossover, the string of the offspring starts by copying the symbols from one parent's string, then at a randomly chosen crossover point begins copying symbols from the corresponding portion of the second parent's string. Similar to 1-point crossover, 2-point crossover randomly selects two crossover points, and at the second crossover point, the offspring's string returns to copying symbols from the corresponding portion of the first parent's string. In uniform crossover, after each symbol position is copied from a parent's string to the offspring's string, there is a fixed probability, $p(c)$, that the parent providing the symbols for the offspring string will be swapped for the other parent [24].

After the crossover process is complete, the next step is to mutate the offspring. The mutation process typically involves changing one or more symbols in the genotype strings. A common method for changing symbols is to have a small but fixed probability of symbol change applied to each position in the string [24, 110]. For binary strings, this is called *bit-flip mutation*, and a typical probability for the mutative flip of an individual bit is $\frac{1}{L}$ where L is the number of bits in the string [24].

In most genetic algorithms, the parent generation is discarded, and all of the offspring are selected to survive as the next generation. Unlike evolutionary programming and evolution strategies, there is no survival selection in genetic algorithms because parent selection alone provides sufficient selection pressure.

2.4 Learning Agents

As mentioned in section 1.2.3, fully autonomic controllers possess self-awareness of their own performance and are able to self-adapt their control procedures. For autonomic controllers employing heuristic search, self-adaptive tuning of the heuristic search is a natural first step toward fully autonomic operation. Some mechanisms for self-adaptive tuning of heuristics have been developed for use in games, compilers, and theorem proving [41, 99, 117, 121, 127].

Pattern classification through supervised learning may support adapative heuristic selection. In general terms, a pattern classifier examines objects and predicts each object's associated class [114]. In Chapter 5, the object to be classified will be an optimization problem. An object is usually represented as a point in an n-dimensional space. Each dimension of the space is called a feature. The features of an optimization problem include utility functions, application design, available architectural patterns, and environmental factors. Classifiers leverage the information in the features and analyze feature interactions to make successful predictions. The classifying techniques described below are used to classify optimization problems according to the most appropriate heuristic for solving the particular problem.

In supervised machine learning, classifiers require a training data set. The training data consists of a set of objects and their corresponding classifications. Most classifiers employ a learning algorithm that studies training sets and makes generalizations [8, 114]. Often the classifiers are tested against a second set of objects called the validation set. The validation set is independent from the training set and is used to ensure that the classifier has not overfit the data [8]. Overfitting occurs when the classifier makes bad generalizations based on noise in the original data set. Finally, a third set of objects (independent from the training and validation sets) called the test set is used to assess the final performance of the classifier [8].

2.4.1 K-Nearest Neighbor

K-Nearest Neighbor (KNN) methods classify an object by examining the k closest objects found in the training set [114]. Each of the k nearest objects *votes* for its own class. The KNN method assigns the class receiving the most votes to the object in question. The training data is the classifier in KNN, so there is no training process [114]. The time complexity of classification in KNN is $O(n)$ where n is the size of the training data set; this is significantly slower than other classification algorithms [114].

Three different distance metrics have been employed in KNN classifiers. The most commonly used is the classic Euclidean distance function. A less computationally expensive distance function is the Manhattan distance function, which is the sum of all feature

differences [114]. A 2-dimensional example of Manhattan distance is the distance a pedestrian must walk between two points in a rectangular city grid such as Manhattan. The Mahalanobis distance function is significantly more expensive computationally than either Euclidean or Manhattan distances but enhances feature independence, which in some cases can improve classification [131]. Recent research into KNN methods has focused on scaling features based on local information in the neighborhood of the object [28, 40].

2.4.2 Support Vector Machines

Support Vector Machines (SVM) belong to the group of classifiers known as linear discriminants. Linear discriminant classifiers find a set of hyperplane decision boundaries to separate the classes. Typically, a hyperplane decision boundary is described by a vector, $\mathbf{w}$, and an offset value, b, and can be used to make a class prediction for an object with features described by $\mathbf{x}$ [8]:

$$y(\mathbf{x}) \;=\; \mathbf{w}^{\mathrm{T}}\mathbf{x} \;+\; b \tag{2.16}$$

If the value of $y(\mathbf{x})$ is positive, then then the classifier predicts that the object represented by $\mathbf{x}$ belongs to the positive class. If the value of $y(\mathbf{x})$ is negative, then the classifier predicts that the object represented by $\mathbf{x}$ belongs to the negative class. If $y(\mathbf{x})$ is equal to 0, then $y(\mathbf{x})$ resides on the decision boundary, and the membership prediction of the object represented by $\mathbf{x}$ will depend upon the implementation of the classifier.

SVMs differ from other discriminant methods by introducing the concept of the margin. The margin is a buffer on either side of the hyperplane decision boundary. In the simplest formulation of SVM, the two classes must be linearly separable, and no points from either class may reside within the margin. Points that are located on the edge of the margin are called *support vectors*. SVM methods seek to find the $\mathbf{w}$ and b that will maximize the margin width. The margin width is equal to the the distance between a point represented by a support vector and the hyperplane decision boundary [8]. This can be computed as follows:

$$\frac{t_n y(\mathbf{x}_n)}{\|w\|} \qquad t_n \in -1, 1 \tag{2.17}$$

where n is the support vector's index in the training set and t_n is the class label for $\mathbf{x}_n$. As seen in Equation 2.17, the width of the margin in SVM is proportional to $\frac{1}{\|w\|}$. To maximize the margin width, SVM minimizes $\|w\|$.

A more sophisticated formulation of SVM allows training on data sets that are not linearly separable. This is accomplished by allowing support vectors to reside within and beyond the margins of the decision boundary. If $\mathbf{x}_n$ is a support vector, then the distance that $\mathbf{x}_n$ has crossed its margin boundary is represented by a slack variable, ξ_n, which is measured in units of margin width. If $\mathbf{x}_n$ is not a support vector, then ξ_n is set to 0. In this more sophisticated formulation of SVM, the goal of maximizing margin width is balanced with the goal of limiting the intrusion of support vectors into the margin. Balancing these two goals can be achieved by minimizing the following expression [8]:

$$C \sum_{n=1}^{N} \xi_n + \frac{1}{2} \|w\|^2 \tag{2.18}$$

where C is an inverted regularization parameter, and N is the number of objects represented in the training set. Regularization parameters allow tuning of the classifier to prevent underfitting and overfitting.

Kernels for SVM

By projecting training sets into higher dimensional spaces, it is often possible to improve the linear separability of the positive and negative classes. A kernel function, $k(\mathbf{x}, \mathbf{x}')$ can be constructed from the dot product of the feature space mapping, ϕ [8]:

$$k(\mathbf{x}, \mathbf{x}') \;=\; \phi(\mathbf{x})^{\mathrm{T}} \phi(\mathbf{x}'). \tag{2.19}$$

The hyperplane decision boundary in Equation 2.16 can be re-formulated using a feature space mapping ϕ:

$$y(\mathbf{x}) \;=\; \mathbf{w}^{\mathrm{T}} \phi(\mathbf{x}) \;+\; b. \tag{2.20}$$

By adding Lagrange multipliers, a Lagrangian function of Equation 2.18 can be developed. From the Lagrangian function, the following expression for $\mathbf{w}$ is developed [8]:

$$\mathbf{w} \;=\; \sum_{n=1}^{N} a_n t_n \phi(\mathbf{x}_n) \tag{2.21}$$

where a_n is a Lagrange multiplier term for $\mathbf{x}_n$. A dual representation of the Lagrangian function can be obtained by substituting this expression for $\mathbf{w}$ back into the Lagrangian function [8]. After this substitution, all instances of ϕ in the dual representation take the form of dot products: $\phi(\mathbf{x}_n)^{\mathrm{T}}\phi(\mathbf{x}_m)$. This allows the substitution of the kernel function, $k(\mathbf{x}_n, \mathbf{x}_m)$, into the dual representation of the Lagrangian function. Equation 2.20 can also be reformulated to use the kernel function [8]:

$$y(\mathbf{x}) \;=\; \sum_{n=1}^{N} a_n t_n k(\mathbf{x}, \mathbf{x}_n) \tag{2.22}$$

Equation 2.22 and the dual form of the Lagrangian function no longer contain ϕ. Thus, it is not necessary to compute $\phi(\mathbf{x})$—ϕ exists and is a valid feature mapping for a given kernel function, $k()$. This concept is referred to as the *kernel trick* [8]. This is useful because kernel functions are often simpler and easier to work with than corresponding feature space mappings.

A number of valid kernels have been developed for use in SVM. Due to its flexibility and power, the radial-basis function is a popular kernel function [8, 54]:

$$k(\mathbf{x}, \mathbf{x}') \;=\; \exp\left(-\gamma \left\|\mathbf{x} - \mathbf{x}'\right\|^2\right), \qquad \gamma > 0 \tag{2.23}$$

This kernel function will be used in some of the meta-controllers described in Chapter 5.

Cross-Validation

When a classifier has been trained, its generalized performance (i.e., prediction accuracy) is initially not known. The performance of the classifier can be determined by testing the classifier against a validation set of the data. The validation set should be independent of the training set. Thus, before training commences, the available data for the classification problem should be split into a training set and a validation set. Determining how to apportion the available data between the training set and the validation set is sometimes difficult. If the training set is too small, it may not provide enough information to produce a useful classifier. If the validation set is too small, the error in the estimated accuracy will be large.

In cases where the amount of available data for the classification problem is small, there may be no reasonable way to split the data into a training set and a validation set. In such cases, the best approach is to employ a technique termed cross-validation. In cross-validation the data set is randomly divided into n equally-sized groups called *folds*. The classifier is then trained and tested through n iterations. With each training/testing iteration, a different fold is held back as the validation set, while the classifier is trained with the remaining $n-1$ folds. When the cross-validation is complete, each problem available to the classifier has been used in a validation set one time. The accuracy of the classifier can be computed by assessing the results on the n validation sets.

Parameter Determination

A challenge in using SVM classifiers is selecting appropriate values for the regularization parameter, C, as well as any parameters associated with the kernel function used. Kernel function parameters influence the classifier's model complexity. Models that are overly complex may risk overfitting the data (i.e. selecting a hyperplane boundary fitted to noisy data that will generalize poorly). Models that are insufficiently complex may risk underfitting the data (i.e. the feature space mapping fails to yield sufficient linear separability). SVM classifiers using poorly selected parameters may suffer considerable performance degradation due to over-fitting or under-fitting the training data.

For kernels that have only one kernel function parameter (e.g., the radial-basis function

kernel described in Equation 2.23), the authors of the `libsvm` library [17] recommend using grid search for optimizing C and the kernel parameter [54]. The `libsvm` library includes a grid search tool for radial-basis functions. Using cross-validation, this tool evaluates classifiers at various points of $(log(C), log(\gamma))$. The accuracy is plotted in topological form as shown Fig. 2.3.

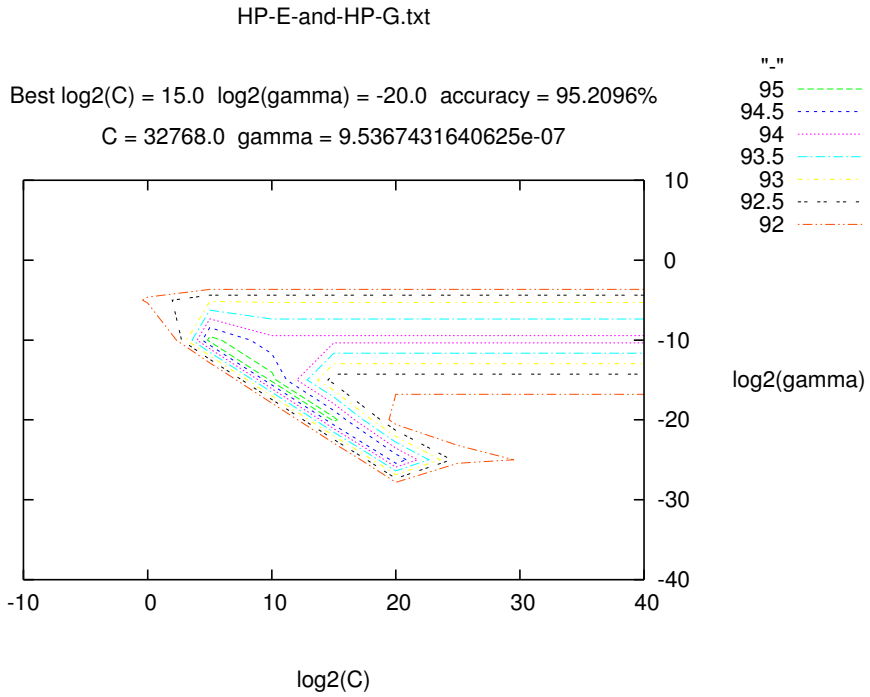


Figure 2.3: Output from grid search tool in `libsvm`.

The grid search tool takes the grid range as an input, so the user may choose to zoom in on the more promising portions of the grid with higher resolution in $log(C)$ and $log(\gamma)$. This process can be repeated until no further improvements in accuracy can be found. Using the $log(C)$ and $log(\gamma)$ SVM parameters that provided the highest accuracy in the grid search, the classifier can then be trained with all available data (i.e. all folds are used vice $n-1$ folds in cross-validations).

2.5 Self-Architecting Software Systems

The proposed autonomic controller will serve as the optimization engine within the SASSY (Self-Architecting Software Systems) framework. This section begins with an overview of SASSY and then presents the framework for enabling autonomic management of SOA applications in SASSY.

2.5.1 Overview of SASSY

SASSY is a model driven framework for run-time self-architecting and re-architecting of composed services and SOA applications [92]. The objective of SASSY is to automate composition, adaptation, and evolution of SOA systems in a way that maximizes global utility over time. The utility functions in SASSY grade the service or application on various QoS metrics and on the incorporation of optional service features.

SASSY uses four types of run-time models: 1) service activity schemas with their corresponding service sequence scenarios, 2) system service architecture models, 3) QoS architectural pattern models, and 4) QoS analytic models [92, 94].

2.5.2 Service Activity Schema (SAS)

SASSY is designed to enable domain experts to build composed services and applications. Through the use of a domain ontology, the domain expert creates a service activity schema (SAS). An SAS contains a process flow that connects incoming requests to sequences of activities and outgoing responses. An activity could be a known service type (from the domain ontology) or another SAS. Requests, activities, and responses can be linked together using gateway constructs reminiscent of Business Process Modeling Notation (BPMN) [77].

Figure 2.4 shows an example of an SAS for an emergency response service in the SASSY user interface [92]. The SASSY user interface offers a modeling language designed to enable domain experts to build SOA applications. The SASSY user interface was built using the Generic Modeling Environment (GME) from Vanderbilt University's Institute for Software Integrated Systems (ISIS) [23]. This SAS represents a service that plans evacuations in response to some natural disaster or security threat. The dashed boxes on the left-hand

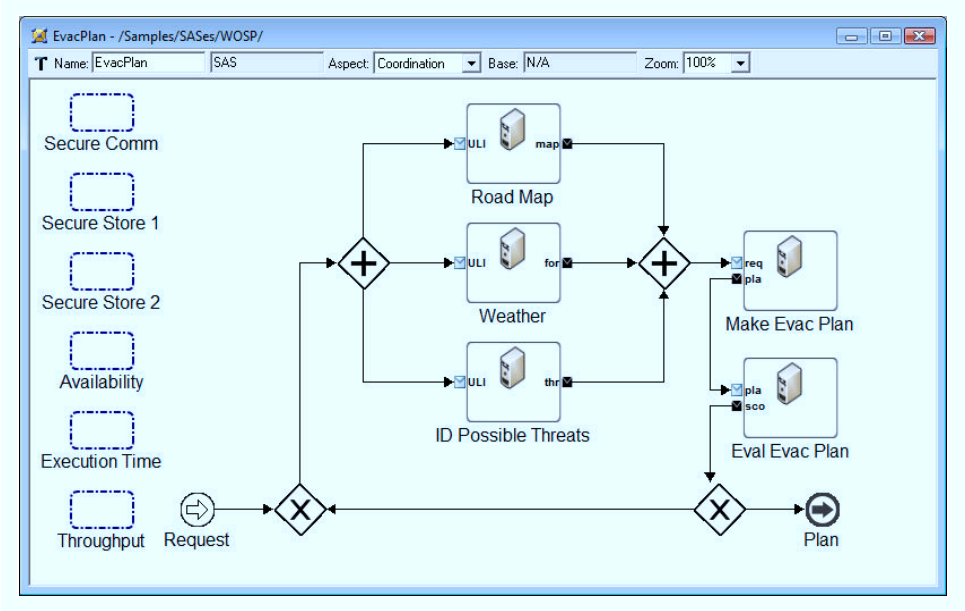


Figure 2.4: Example of a service activity schema (SAS) for an evacuation planning service [92].

side of Fig. 2.4 correspond to service sequence scenarios (SSS) (see Section 2.5.3). This evacuation planning service combines five composable service types (see boxes with rounded corners and a server icon inside): `Road Map`, `Weather`, `ID Possible Threats`, `Make Evac Plan`, and `Eval Evac Plan`. When a message containing a plan request is received by the evacuation service, three service types are invoked simultaneously (see fork gateway in the diagram specified by a rhombus with a plus sign inside): `Road Map`, `Weather`, and `ID Possible Threats`. The `Road Map` service type is used to obtain machine readable maps of the roads in the region. The `Weather` service type provides current and predicted weather information about the region and the `ID Possible Threats` service type identifies possible threats for the region and assigns probabilities to each threat. After all three activities complete, i.e., join, the `Make Evac Plan` service type is invoked to generate an evacuation plan, which is passed to the `Eval Evac Plan` service type for an analysis and evaluation of the evacuation plan [92]. The result of the evaluation is tested at the conditional gateway (see rhombus with an "×" in the middle at the output of `Eval Evac Plan`). If the plan is not approved, a new one will have to be produced. If the plan is approved, a response

message containing the plan is sent back [92].

2.5.3 Service Sequence Scenario (SSS)

Domain experts express QoS requirements through SSS modeling constructs. For each SSS, the domain expert selects a subset of the service types from the SAS; these selected service types must 1) define a fully connected complete or partial path through the SAS, 2) follow only one path through a switch, and 3) follow all paths through a fork-and-join [77]. Once the service types of the SSS are selected, the domain expert selects one QoS metric defined in the SASSY ontology to associate with the SSS [92]. An attribute utility function operating on the value (or possibly distribution) of the QoS metric is also associated with each SSS (see Section 2.2.3). Utility functions are established by the domain experts in consultation with all stakeholders [92].

Figure 2.5 displays three SSSs: the `Execution Time` SSS, the `Availability` SSS, and the `Secure Comm` SSS. The `Execution Time` SSS includes the fork-and-join of `Road Map`, `Weather`, and `ID Possible Threats` followed by the execution of `Make Eval Plan` and `Eval Evac Plan`. The QoS metric associated with this SSS is execution time. The `Availability` SSS has the same structure as the `Execution Time` SSS except that its metric is availability. Thus, two or more SSSs may have the same structure as long as they have different metrics associated with them [92]. The `SecureComm` SSS includes the connection between `ID Possible Threats`, `Make Eval Plan`, and `Eval Evac Plan`. The metric associated with this SSS is the security level of its communication [92].

2.5.4 System Service Architecture (SSA)

A System Service Architecture (SSA) consists of a structural model and a behavioral model of an SOA system [92]. Unlike traditional software architectural models that are primarily used during the design phase, the SASSY framework uses the SSA as its representation of the running system [94]. Evolution and adaptation of SASSY systems are enabled through run-time reasoning and analysis of the SSA. The SSA's structural model is based on the eXtensible Architectural Description Language (xADL) [21], which provides the traditional

component-and-connector view of the software architecture [92].

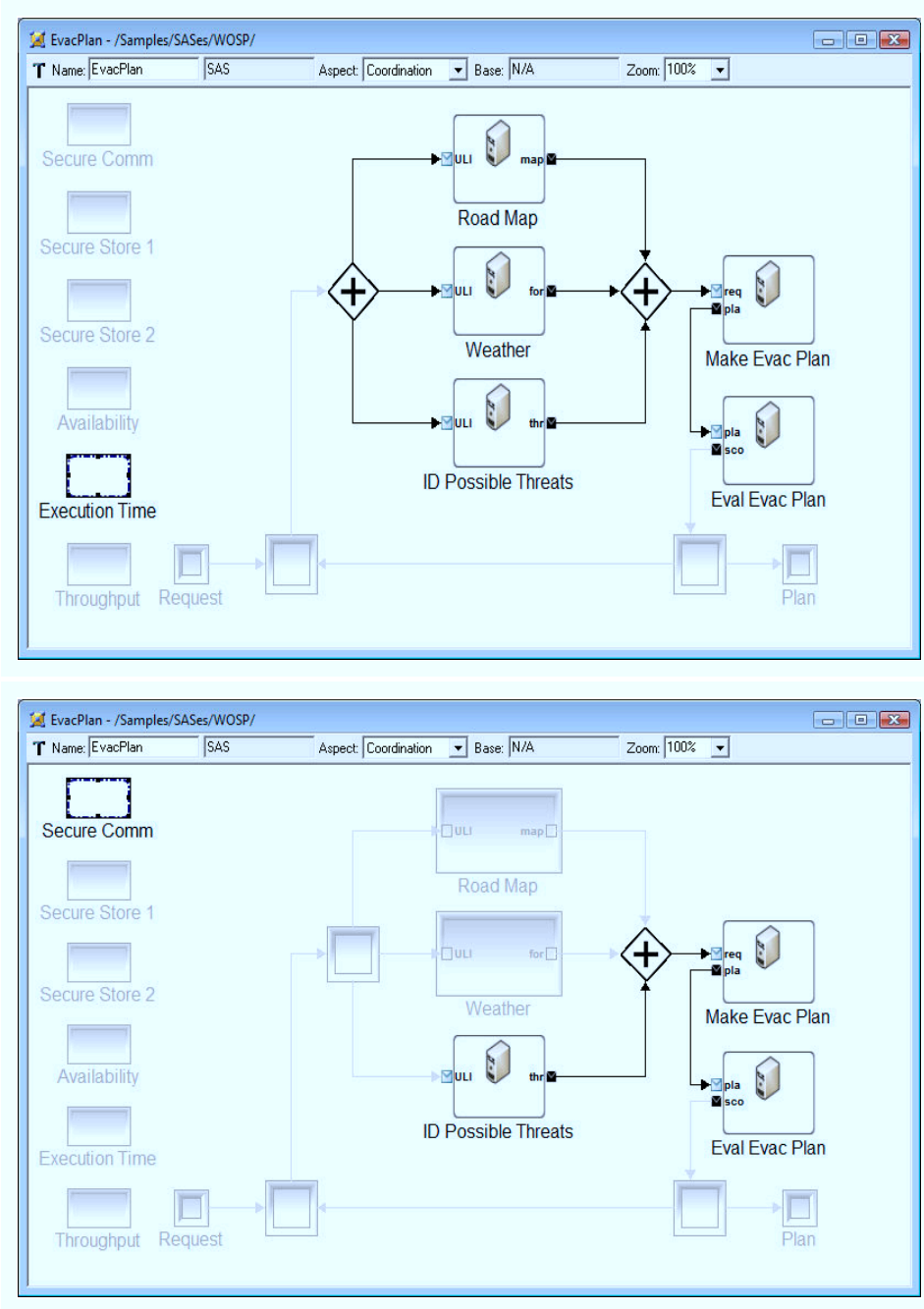


Figure 2.5: Top: `Execution Time` or `Availability` SSS (they have the same structure); Bottom: `Secure Comm` SSS [92]

The SSA extends the core xADL language by introducing the concept of service instances, which are modeled as software components or a composition of software components. A *service instance* is the realization of a service type defined in the ontology [92]. The SSA also provides a mapping between each service instance and the concrete service provider that is used at a given moment [92]. The middleware enabling integration and communication among the services is modeled as software connectors [92]. Finally, the components and connectors bind to one another using required and provided interfaces [92].

The top part of Fig. 2.6 shows the structural view of a base SSA that corresponds to the SAS of Fig. 2.4 [92]. The base SSA is automatically generated by SASSY using GReAT [2, 3], which is a graph transformation tool [92]. The base architecture associates one component to each service type and creates one component to represent the logic of a coordinator that orchestrates the communication among service types [92]. The bottom part of Fig. 2.6 shows a modified version of that base SSA in which the component `Eval Evac Plan` has been replaced by a fault-tolerant component [92].

The SSA's behavioral models provide a state machine view for representing how the service instances interact with one another as they fulfill the system requirements [92]. In other words, a behavioral model corresponds to the executable logic of service coordination in SOA [92]. An SSA's behavioral models are based on Finite State Processes (FSP) [36, 68], which unlike many other state machine languages provides a rich set of abstraction constructs, enabling scalable behavior modeling of large-scale software systems [92]. SASSY automatically generates the SSA's behavioral models based on the requirements specified by the domain expert in the SAS [92]. Since the SSA contains both structural and behavioral models, the SSA needs only a set of selected service providers and bindings with those services to become executable [92].

2.5.5 Architectural Patterns

The SASSY framework uses a library of *architectural patterns* to assist in the self-architecting process [92, 119]. Each pattern consists of one or more components which may be linked by connectors. Each of these components may be associated with one or more service types, which may be instantiated by multiple service providers. The structural and

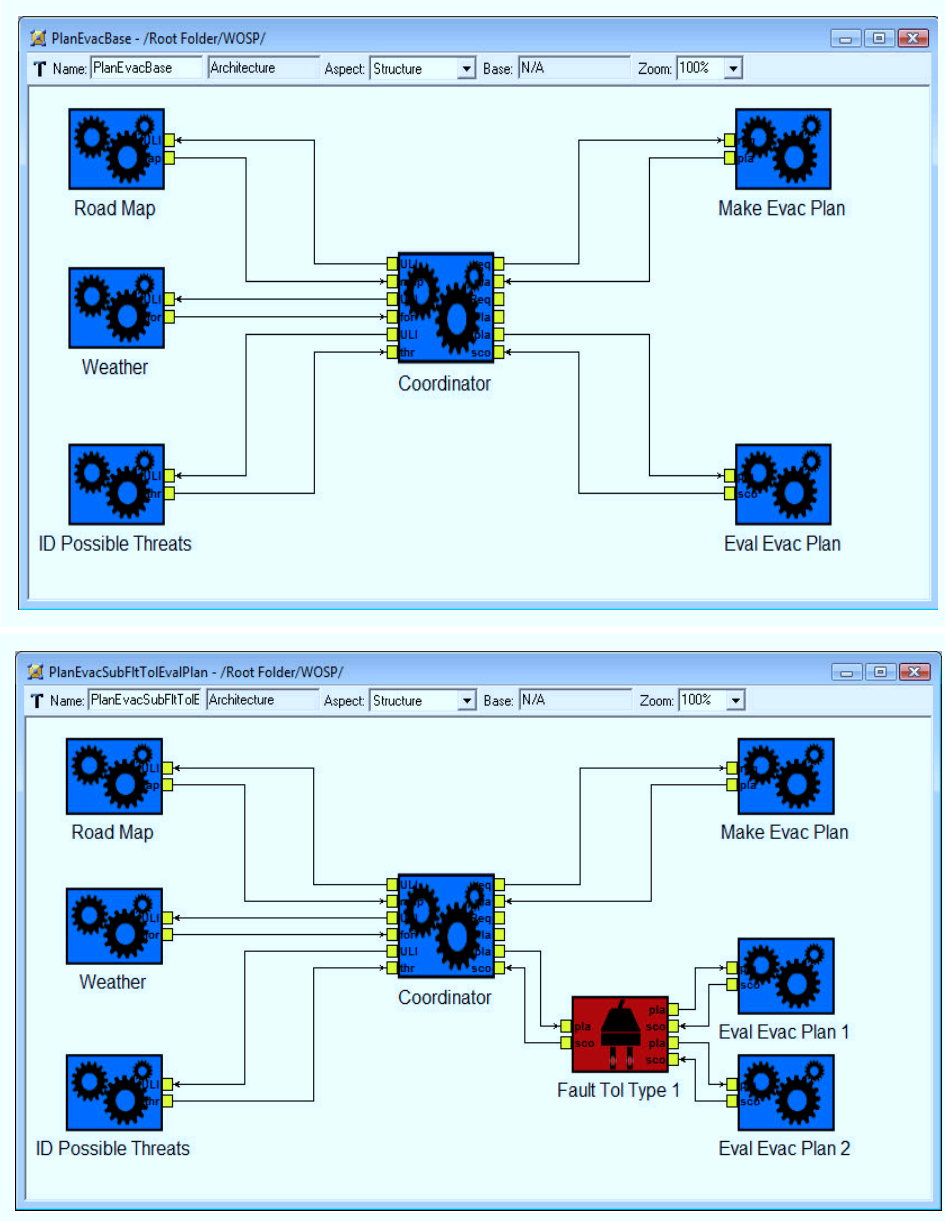


Figure 2.6: Top: Base SSA corresponding to the SAS of Fig. 2.4; Bottom: SSA showing the replacement of component `Eval Evac Plan` with a fault tolerant component [92].

behavioral aspect of a pattern are described in xADL [21] and FSP [68], respectively [92,94]. A pattern is also associated with QoS metrics where the pattern may provide tangible improvements [92].

The class diagram in Fig. 2.7 illustrates the relationships between patterns, SSSs, components, service types, QoS metrics, and utility functions [92]. Each SSS has a single utility function associated with it. Each utility function is a function of a single QoS metric. Each QoS metric can be computed by an analytic QoS model (see Section 2.2.2).

An example of an architectural pattern is the fast fault tolerant pattern shown in the bottom of Fig. 2.6 [92]. The fast fault tolerant pattern influences two QoS metrics: availability and execution time [92]. When the fast fault tolerant pattern receives a service request from the coordinator, the fast fault tolerant pattern forwards a copy of the service request to each of its connected service instances. The fast fault tolerant pattern returns the first service response back to the coordinator and ignores the subsequent response from the other service instances. For each pattern, a set of analytic QoS models can be developed [94]. The example below shows simple analytic models for the fast fault tolerant pattern in Fig. 2.6 [92]:

$$A = 1 - (1 - a_1)(1 - a_2) \tag{2.24}$$

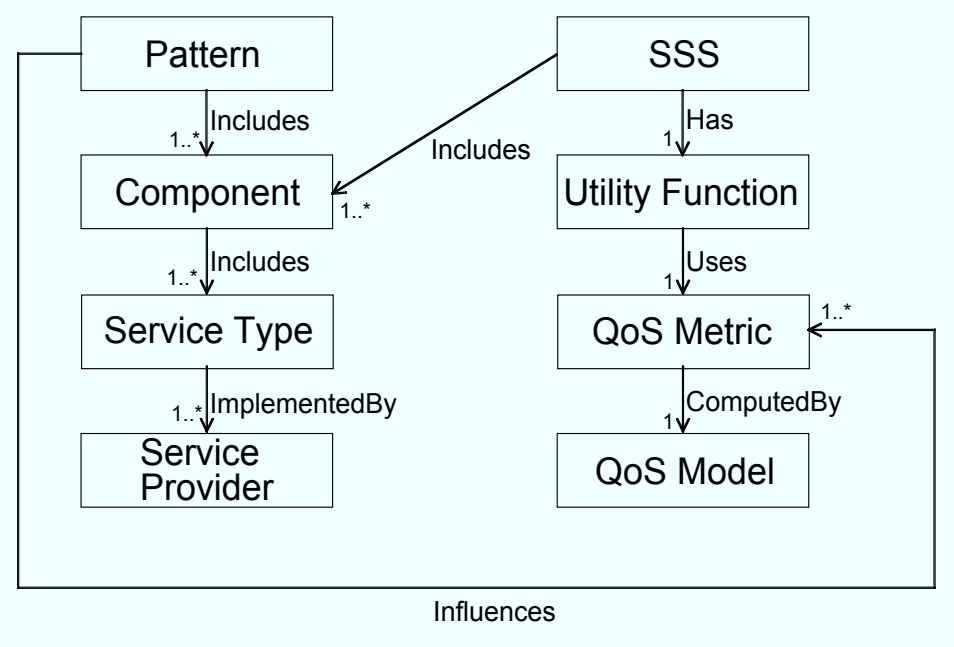


Figure 2.7: Class diagram for SSSs [92].

assuming failure independence, and

$$E = \frac{a_1(1-a_2)}{A}e_1 + \frac{a_2(1-a_1)}{A}e_2 + \frac{a_1 a_2}{A}\min\{e_1, e_2\} \tag{2.25}$$

where e_1 and e_2 are the execution times of the individual components, A is the availability, and E is the execution time. More detailed models for this pattern and others are provided in [94] and can be developed from [88].

2.5.6 Performance Models for Architectural Patterns

The following general rules, when applied recursively, can be used to obtain the SSS QoS models for availability, execution time, and throughput [92].

- Availability: the availability of a sequence is the product of the availabilities of the components in the sequence. The availability of a fork-and-join is the product of the availabilities of all of its branches.
- Execution time: the execution time of a sequence is the sum of the execution times of the components in the sequence. The execution time of a fork-and-join is the maximum execution time of each of its branches.
- Throughput: the throughput of a sequence is the minimum throughput of the components in the sequence. The throughput of a fork-and-join is the minimum throughput of its branches.

2.5.7 Three Layer Model of SASSY

As discussed previously, SASSY uses the Kramer-Magee 3-layer model [69] as the overall structure for the autonomic system. The three layers are the Goal Management Layer, the Change Management Layer, and the Component Control Layer.

Component Control Layer in SASSY

The Component Control Layer in SASSY provides the interface between the SASSY system and individual service providers (SPs). The Component Control Layer receives instructions

from the Change Management Layer to connect specific SPs to the SASSY system or to disconnect specific SPs from the SASSY system. To accomplish this task, the SASSY Component Control Layer employs adaptation connectors that implement SOA adaptation patterns [45–47, 49].

The Component Control Layer also performs low-level monitoring of the component and SP performance. The collected information is passed up to the Change Management Layer [43].

Change Management Layer in SASSY

The Change Management Layer receives metrics and status updates from the Component Control Layer. A performance monitor or gauge service within the Change Management Layer uses this information to assess the system's current global utility, U_g.

If the system's U_g falls below some threshold or a new resource becomes available, the Change Management Layer may request a new plan from the Goal Management Layer. In SASSY, the plan is a tuple, $(\mathcal{A}, Z)$, of a new architecture, $\mathcal{A}$, and a new service provider selection, Z.

The Change Management Layer has the responsibility of implementing the plans (i.e., new $\mathcal{A}$ and Z) it receives from the Goal Management Layer [43].

Goal Management Layer in SASSY

In SASSY, the Goal Management Layer's main function is to determine a new $\mathcal{A}$ and accompanying Z. When the Goal Management Layer receives a new plan request, the Goal Management Layer undertakes a re-architecting process that finds a new $\mathcal{A}$ and Z. When the re-architecting process completes, the Goal Management Layer then sends the new $\mathcal{A}$ and Z to the Change Management Layer. The work presented in Chapters 4 and 5 performs the re-architecting process for the SASSY system.

2.5.8 Definitions

This section provides formal defintions of some of the SASSY constructs previously discusssed. These definitions are not meant to define a new software architectural description

language [56] but to establish the concepts required 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 2.8 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.

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

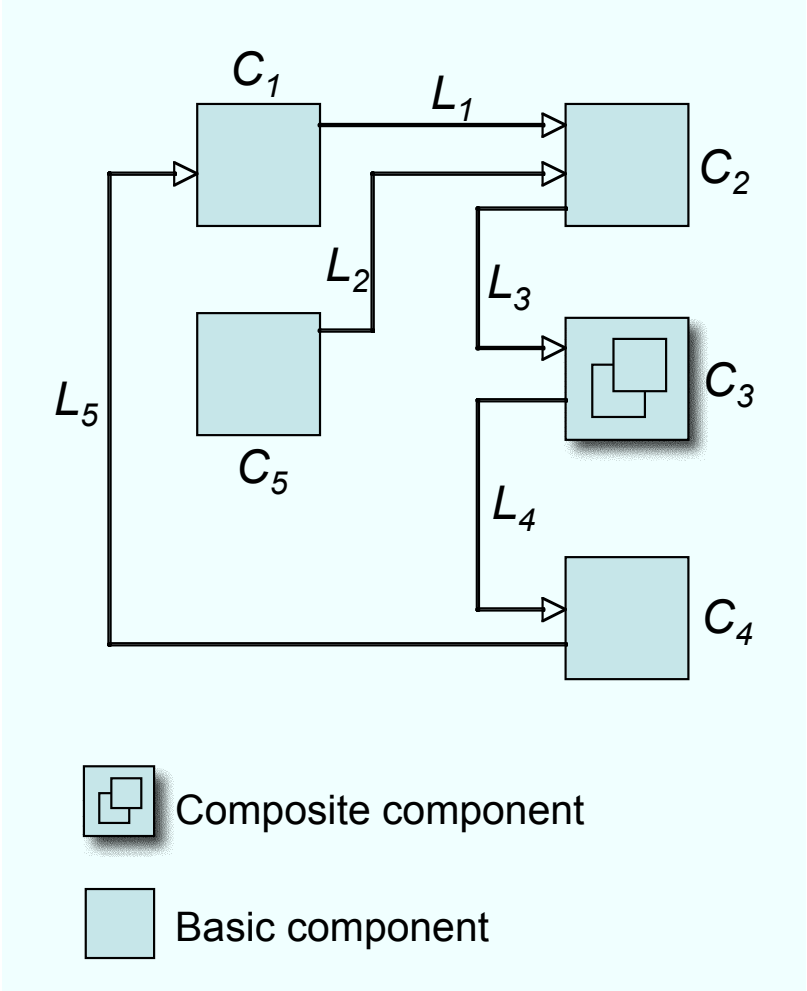


Figure 2.8: Depiction of an architecture.

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.

2.6 Related Work

QoS brokers try to match the requirements of service consumers with the capabilities of SPs [90, 93]. The problem of optimal service selection has been studied in [13, 16, 30, 87, 88]. The J-Opera software package offers the ability to automatically compose services and applications and offers some limited opportunities for optimization[50, 103, 104].

Frameworks for the evolution and adaptation of software systems have been described in [18, 22, 44, 66, 80]. Some research such as [105, 106, 115] has focused on self-healing processes.

Optimization of the placement of components, services, and applications is a major research initiative in the mobile and ubiquitous computing communities [78, 79, 97, 101, 108, 118]. The authors in [4, 15, 83] study optimization of both individual and composed services.

In [12], 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 [14]. 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 [81] 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 [67], employs architectural tactics in a multi-objective evolutionary algorithm to expedite the multi-objective search process. Martens et al. present a similar system in [82] that starts quickly by using LP on a simplified version of the problem to prepare a starting population for a multi-objective evolutionary algorithm.

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 [33, 34].

Chapter 3: The Need for Meta-Controllers in Autonomic Computing

3.1 Challenges of Autonomic Controllers

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 to manage complex systems.

Run-time self-optimization in the face of changes in the environment presents special challenges. 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 [14, 39, 124].

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.

3.1.1 Tuning Optimization Algorithms

The behavior of optimization algorithms is often controlled by a number of parameters. Typically, the level of sophistication in the optimization algorithm is correlated with the number of parameters guiding the algorithm's behavior. Tuning the parameters that guide optimization algorithm behavior to reasonable settings requires significant expertise and likely substantial effort on behalf of a human administrator.

An examination of the heuristic search algorithms presented in Section 2.3.3 illustrates a number of control parameters in optimization algorithms that require the attention of system designers and administrators. For each of the heuristic search algorithms considered below, the budget for the search (either measured in time or number of evaluations) is also sometimes a tunable parameter.

The behavior of a hill-climbing heuristic search algorithm is impacted significantly by the selection of its mode of operation, either greedy or opportunistic. On certain problems, hill-climbing algorithms may employ some form of neighborhood filtering. A neighborhood filter may be defined by multiple parameters specific to the domain of the optimization problem. Performance of a hill-climbing algorithm is likely to be impacted by interactions between the mode of operation and the settings of the neighborhood filter.

Like hill-climbing, beam search may also employ neighborhood filters. Additionally, a beam search algorithm must use one of three criteria levels for determining the next level-list. Beam search also requires a setting for the the beam width. Each of these parameter settings can have significant impact on a beam search algorithm's performance. Interactions of these parameter settings are likely to have a performance impact as well.

Simulated annealing requires two parameters, the initial and final probabilities of moving

to an inferior neighbor. In some cases, the definition of an inferior neighbor must also be set. The correct settings for these probabilities will depend considerably on the search budget and the utility topology of the search space.

Evolution strategies and evolutionary programming require a large number of parameter settings:

- parent population size,
- offspring population size,
- overlapping population boolean,
- method of offspring generation,
- mutation step sizes (one for each dimension of the solution being mutated), and
- adaptive step factors (one for each dimension of the solution being mutated),

All of these parameters can be expected to have significant impacts on the evolutionary algorithm's performance. Heavy interaction between many of the parameters should also be expected to affect the evolutionary algorithm's performance.

3.1.2 Effect of Human Effort on Cost

With some trial and error, it may be possible for a human system administrator with experience in heuristic search optimization to determine reasonable parameter settings for a given heuristic search optimization algorithm. Such expertise is likely uncommon among system administrators, and devoting time and effort to this task would reduce the administrator's available effort in other areas. Determining optimal parameter settings for heuristic search optimization algorithms, however, could require a major research project analyzing the autonomic system and its environment.

Further, changes in the autonomic system either in scale or in function (e.g. a significant increase in the number of users or the introduction of new functionality) could alter the optimal parameter settings for the heuristic search algorithms, thus requiring an investment of time and resources by the human administrator to re-assess the system. Changes in the

environment (e.g. changes in the provisioning of hardware available to the system) could have the same effect.

Given the myriad ways that the parameter settings and the autonomic controller may need to be reassessed, it can be expected that the costs of managing such systems could be high, which would increase the TCO—reducing TCO is one of the fundamental goals of autonomic computing. Chapters 4 and 5 will present automated methods that can substantially reduce the human effort required and attendent costs.

3.2 Case Study: Autonomic Load-Balancing

As mentioned in Section 2.1.3, an autonomic controller employing a hill-climbing heuristic search has been applied to optimizing business load-balancing [37]. Further work (currently unpublished) on autonomic load-balancing has explored using beam search, simulated annealing, evolutionary programming, evolution strategies, genetic algorithms, and particle swarm. Over two hundred and fifty different heuristic parameter combinations were tested against the fifty dynamic stress tests developed in [37]. In the unpublished heuristic study, the fifty dynamic load stress tests contained substantial variance in global utility potentials, although individual utility landscapes representing snapshots in time were relatively smooth and simple. Also, utility predictions of load balancing policies were expensive, limiting the evaluation budget. As a result, hill-climbing and beam search outperformed the other tested heuristics; this was determined according to the analytic methodology described below. The autonomic controller described in Chapter 4 is expected to encounter landscapes with substantially more variation than those in the unpublished work; also, the changes in architectural design and service selection are expected to be considerably less granular than modifications to load-balancing and resource allocation policies. Therefore, more sophisticated heuristics may be employed by the autonomic controller in Chapter 4.

3.2.1 Overview of Autonomic Load-Balancing

Load balancing refers to a number of widely used techniques for distributing work among multiple resources according to a given policy. In recent work, autonomic principles have

been applied to the development of dynamic load balancing policies that allow system adaptation in the face of an uncertain and changing environment [9, 11, 73]. Some dynamic load balancing policies seek to improve system efficiency by dispatching a work request to a specific resource where the effort required to process the request is minimized or where service level objectives are most likely to be met [9, 11, 26, 73, 132]. Other dynamic load balancing policies seek to prioritize work requests that generate more utility [107]. This work uses some of the dynamic load balancing policies first described in [85] that prioritize the requests most likely to generate utility. The autonomic controller allows for greater precision in the development of these load balancing policies, provides the capability to reallocate cluster resources, and is well-suited for highly dynamic workloads.

This work provides a *business-oriented* approach to dispatching incoming requests to servers and allocating servers to server clusters according to customer priority classes. This approach maximizes utility across multiple levels of offered loads and adapts and reacts well to highly dynamic loads and can allocate resources among user classes in two different ways.

The approach considered in this chapter is aimed at improving the revenue of an e-commerce site, an auction site in this example, by providing better performance to groups of customers that have higher business value at the expense of other less important customers.

3.2.2 Optimization Problem to be Solved

The autonomic LB optimizes a global utility function that calculates the business-value generated by the throughput of specific revenue-generating transactions with a certain expected response time. The *bid* transaction throughput is used because it generates revenue for the bidding site. Good response times are also critical to generating value—when response times are good, current customers continue to use the auction site and new customers are attracted to the site by favorable impressions. If response times are poor, customers are likely to abandon the site and use a competing auction site with better response times—this effect deprives the site of future business. This work uses the sigmoid utility function $U_s^R(R_s)$ from [7] as the response time factor in the global utility function. This response time utility function, shown in Eq. (3.1), models whether utility is generated by complying

with the response time service level objective (SLO):

$$U_s^R(R_s) = \frac{e^{-R_s+\beta_s}}{1+e^{-R_s+\beta_s}} \tag{3.1}$$

where s is the priority class (i.e., gold, silver, or bronze), β_s is the average response time SLO, in seconds, for class s, and R_s is the average response time, in seconds, of all class s transactions. The response time utility function has a value between zero and 1 and goes to zero as the response time goes to infinity. The value of the utility is 0.5 when the response time meets the SLO (i.e., $R_s = \beta_s$). The load balancing policies should maximize the values of U_s^R in a way that prioritizes those customer classes that are most likely to generate bids. This is achieved by combining the response time utility factors in a weighted sum:

$$U_{total}^R(\vec{R}) = \sum_{\forall\ s} w_s \times U_s^R(R_s) \tag{3.2}$$

where w_s are weights defined by management to indicate the priority of class s, and $\vec{R}$ stands for the vector of average response times for the classes.

The *bid* throughput, $X_{bid,s}$, for priority class s is the second factor in the utility. The total bid throughput, X_{bid}, is computed as follows:

$$X_{bid}(\vec{X}) = \sum_{\forall\ s} X_{bid,s} \tag{3.3}$$

where $\vec{X}$ stands for the vector of average bid throughputs for the classes. From a business perspective, $X_{bid}(\vec{X})$ represents how much money is being made today, while $U_{total}^R(\vec{R})$ represents the likelihood of customers returning tomorrow. The goal is to maximize both $X_{bid}(\vec{X})$ and $U_{total}^R(\vec{R})$ in a way that maximizes revenue today while ensuring the most important customers are satisfied and will continue using the site in the future. Making the

global utility, U_g, the product of $X_{bid}(\vec{X})$ and $U^R_{total}(\vec{R})$ achieves this goal:

$$U_g(\vec{R}, \vec{X}) = X_{bid}(\vec{X}) \times U^R_{total}(\vec{R}). \tag{3.4}$$

It should be noted that the values of $\vec{R}$ and $\vec{X}$ depend on the specific policy vector $\vec{s}$ used by the LB (see next section) and on the workload intensity $\mathcal{W}$. Thus, the utility function can be written as a function h of $\vec{R}$, $\vec{X}$, $\vec{s}$, and $\mathcal{W}$ as

$$U_g(\vec{R}, \vec{X}) = h(\vec{R}, \vec{X}, \vec{s}, \mathcal{W}). \tag{3.5}$$

LB Policies

The LB uses two autonomic policies. The first, called *f-policy*, is a re-direction policy that affects the dispatching of requests to server clusters. This policy is specified by three parameters of the form $f_{i,j} \in [-1, 1]$ that indicate the fraction of requests from priority class i to cluster j. In this case, these parameters are $f_{S,G}$, $f_{B,S}$, and $f_{B,G}$. A positive value for $f_{i,j}$ indicates redirection of class i requests to cluster j and a negative value indicates a redirection in the opposite direction.

An f-policy is then characterized by the vector $\vec{f} = (f_{S,G}, f_{B,S}, f_{B,G})$. The autonomic LB dynamically adjusts the f-policy $\vec{f}$ to maximize the global utility U_g.

In order to preserve the identity of the clusters, one more constraint is added to the values within $\vec{f}$: $f_{S,G}$ and $f_{B,G}$ must carry the same sign (i.e., $f_{S,G} \times f_{B,G} \geq 0$). Without this restriction, the autonomic controller will occasionally swap the clusters (e.g., move gold to silver, silver to bronze, and bronze to gold). At first glance, this behavior might seem acceptable, however, moving cluster could result in a number of high priority requests being stuck behind a large number of low priority requests from the previous occupant of the cluster. This ultimately yields unacceptable response times for high priority requests.

The second policy, called *s-policy*, is a resource allocation policy that determines how many servers should be allocated to each cluster. Specifically, it determines the values of n_G, n_S, and n_B, which combined with the values of $f_{S,G}, f_{B,S}$, and $f_{B,G}$, maximize the

global utility function. The s-policy is characterized by the vector $\vec{n}_A = (n_G, n_S, n_B)$.

The state space $\mathcal{S}$ of all possible configurations is formally described below with the help of the $\epsilon(x)$ function defined as 0 for $x \geq 0$ and 1 for $x < 0$.

$$\begin{aligned}
\mathcal{S} \;=\; & \{\vec{s} = (f_{S,G}, f_{B,S}, f_{B,G}, n_G, n_S, n_B) \mid \\
& n_G, n_S, n_B \in \{1, 2, \cdots, N_A - 2\}, \\
& n_G + n_S + n_B = N_A, \\
& f_{S,G}, f_{B,S}, f_{B,G} \in [-1, 1], \\
& f_{S,G} \times f_{B,G} \geq 0, \\
& \epsilon(f_{S,G})|f_{S,G}| + \epsilon(f_{B,G})|f_{B,G}| \leq 1, \\
& (1 - \epsilon(f_{S,G}))f_{S,G} + \epsilon(f_{B,S})|f_{B,S}| \leq 1, \\
& (1 - \epsilon(f_{B,S}))f_{B,S} + (1 - \epsilon(f_{B,G}))f_{B,G} \leq 1\}.
\end{aligned}$$

The three last constraints specify that no more than 100% of the requests initially directed to one cluster can be redirected.

The problem to be solved by the autonomic LB can be cast as the following non-linear constrained optimization problem:

Find the policy vector $\vec{s}_{\max}$ such that $\vec{s}_{\max} = \arg\max_{\vec{s} \in \mathcal{S}} \; \{U_g(\vec{R}, \vec{X}) = h(\vec{R}, \vec{X}, \vec{s}, \mathcal{W})\}$. Note that the function h is non-linear and does not have a closed form expression because the response time and throughput values have to be determined by solving a multiclass closed queueing network model. Although no closed form expression exists for solving multiclass closed queueing networks [86], solutions can be found through iterative or recursive algorithms. Moreover, the state space $\mathcal{S}$ is typically very large. Therefore, using standard optimization techniques is not an option for an autonomic controller that needs to make real-time policy change decisions. For that reason, an autonomic LB that uses heuristic techniques is employed.

The LB controller algorithm considers that time is divided into 30-second time intervals called *controller intervals (CI)*. Two control levels are implemented: 1) the f-policy is re-evaluated at the end of each CI, and the s-policy is re-evaluated by the controller at the

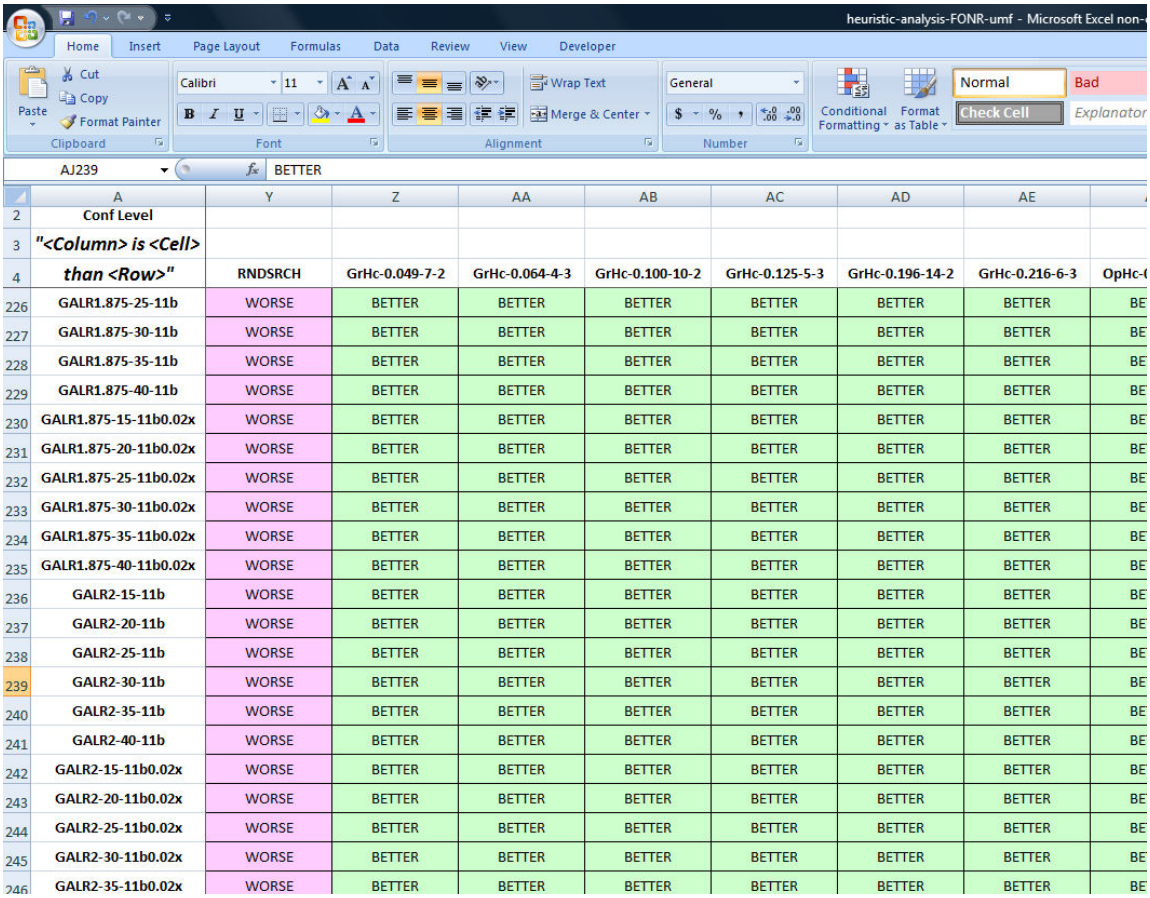
end of every 10 CIs. Because of the switching cost of moving servers from one cluster to the other, the s-policy should be evaluated at a lower frequency than the f-policy.

Each server manages its own queue of requests. The requests in each queue are ordered by the timestamp of the original request such that the oldest request in the queue is the next to be serviced. The LB redirects a fraction of incoming requests from one cluster to another cluster according to the f-policy. The LB sends requests to servers within a cluster in a round-robin fashion. When the heuristic search completes, the autonomic controller may need to move one or more servers from one cluster to another to comply with a new s-policy. To move servers between clusters, the controller undertakes the following actions: 1) an empty, temporary list for storing requests is established for each cluster donating or receiving a server 2) before each server is moved, the requests in that server's queue are transferred to the temporary list of the donating cluster 3) within a receiving cluster, all requests in server queues are transferred to the temporary list for that receiving cluster 4) the servers are moved to comply with the new s-policy 5) the requests in the temporary lists are distributed round-robin to the server queues within each cluster.

3.2.3 Testing Optimization Algorithms

A methodology was developed to reduce the variance introduced by the tests, making statistically significant comparisons of the utility achieved with different heuristics possible. A tool was developed in Microsoft Excel that automatically applies the paired observations variance reduction technique [59] to generate a matrix of comparisons between different heuristics. Figure 3.1 shows a screenshot of a portion of the matrix produced by the tool. This variance reduction tool showed at a 99% confidence level that autonomic controllers using hill-climbing and beam search generated a higher average global utility than controllers using other search heuristics.

The unpublished work demonstrates that paired observation variance reduction is also useful for comparing heuristic search trajectories. A separate tool was developed in Excel to generate automatic plots using paired observations. This plot generation tool compared two heuristics to a reference heuristic, enabling statistically meaningful visualization of mean search trajectories. Such plots are useful in assessing how changes in the evaluation budget



	A	Y	Z	AA	AB	AC	AD	AE
2	Conf Level							
3	"<Column> is <Cell>							
4	than <Row>"	RNDSRCH	GrHc-0.049-7-2	GrHc-0.064-4-3	GrHc-0.100-10-2	GrHc-0.125-5-3	GrHc-0.196-14-2	GrHc-0.216-6-3
226	GALR1.875-25-11b	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
227	GALR1.875-30-11b	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
228	GALR1.875-35-11b	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
229	GALR1.875-40-11b	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
230	GALR1.875-15-11b0.02x	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
231	GALR1.875-20-11b0.02x	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
232	GALR1.875-25-11b0.02x	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
233	GALR1.875-30-11b0.02x	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
234	GALR1.875-35-11b0.02x	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
235	GALR1.875-40-11b0.02x	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
236	GALR2-15-11b	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
237	GALR2-20-11b	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
238	GALR2-25-11b	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
239	GALR2-30-11b	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
240	GALR2-35-11b	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
241	GALR2-40-11b	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
242	GALR2-15-11b0.02x	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
243	GALR2-20-11b0.02x	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
244	GALR2-25-11b0.02x	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
245	GALR2-30-11b0.02x	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER
246	GALR2-35-11b0.02x	WORSE	BETTER	BETTER	BETTER	BETTER	BETTER	BETTER

Figure 3.1: Screenshot of the heuristic comparison matrix for autonomic load balancer.

might impact different heuristics. Figure 3.2 shows a sample plot of an evolution strategy algorithm and a genetic algorithm using random search as a reference.

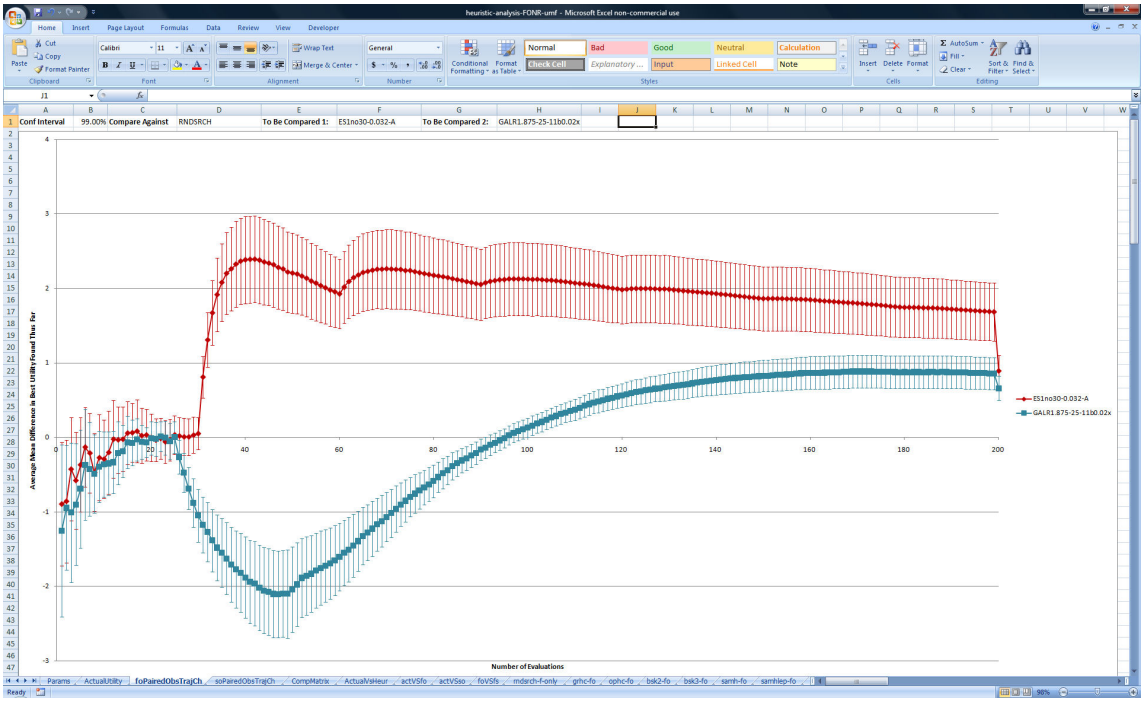


Figure 3.2: Plot of the difference in heuristic search trajectory between random search and an evolution strategy algorithm ($M = 1$, $K = 30$, step size=0.032) and the difference between random search and a genetic algorithm ($M = 25$, linear rank $\mathcal{S} = 0.875$, uni. crossover=0.02, genome=11bits) with 99% confidence intervals.

Overhead of Tuning Optimization Algorithms

The tool described in Section 3.2.3 is useful for making comparisons betweeen heuristic search algorithms. However, it requires a large amount of experimental data to be collected, processed, and inserted into the tool. In the case of the autonomic load balancer, generating and collecting the data took thousands of hours of computation. Further a human administrator had to evaluate the comparisons and make decisions about which heuristic search algorithm and parameters should be evaluated next. Eventually, the human administrator will start encountering diminishing returns in tuning the heuristic search algorithm. This

process can take months of the administrator's time.

3.3 Concluding Remarks

As autonomic systems are developed and deployed, the burden placed on administrators for their maintenance and tuning needs to be carefully considered. Technologies that allow the autonomic system to reflect on its own performance and conduct self-tuning need to be explored and utilized if possible. Revisiting the levels of autonomicity described in Section 1.2.3, adding self-tuning and self-management capabilties within the autonomic controller itself may raise a system from being *autonomous* to being *autonomic*.

Chapter 4: Software Architecture Optimization Search

This chapter provides a comprehensive examination of the optimization conducted by the SASSY autonomic controller. The first section provides the context in which architecture search is performed in SASSY. The second section describes the optimization problem that needs to be solved by the SASSY autonomic controller. The optimization procedures used in this work depend upon performance modeling, so procedures for automated generation of SOA performance models are presented in the third section of the chapter. The proposed framework for solving SASSY's optimization problems is outlined in the fourth section. The fifth section contains heuristic search algorithms that have been adapted to work within the framework. The sixth section examines whether the selection of heuristic search algorithms has a significant performance impact. A meta-optimization framework for autonomic systems is detailed in the seventh section of this chapter.

4.1 Architecture Search Overview

As described in Chapter 2, in SASSY, a domain expert specifies data flows among activities for a new SOA application using a visual language. The domain expert can specify multiple QoS requirements, which are then expressed as SSSes and attribute utility functions. 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, $U_g()$, that combines the attribute utility functions.

When the system's 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 procedures described in Section 4.3.

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 [94]. 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 [43]. This enables the system to re-architect at run-time when new SPs become available or an SP currently bound to the architecture fails.

4.2 The Software Architecture Optimization Problem

The architecture optimization problem in SASSY is to find an architecture and a set of SPs that implement the service types in the SAS in a way that optimizes the SAS global utility function $U_g()$.

More formally, the optimization problem can be expressed as:

Find an architecture A^* and a corresponding service provider allocation Z^* such that

$$(A^*, Z^*) = \text{argmax}_{(A,Z)}\ U_g(A, Z). \tag{4.1}$$

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.

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

$O((p \times s)^n)$. It is clear that the solution space is huge even for small values of p, s and n. 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! In fact, the problem is NP-hard. This chapter presents an optimization technique to avoid an exhaustive, costly, and computationally infeasible search.

4.3 SSS Performance Models

After the SASSY autonomic controller finishes generating the base architecture as described in Section 4.1, the autonomic controller produces a set of SSS performance models; one SSS performance model is created for each SSS. A performance model for SSS i, $\mathcal{M}_i$, in SASSY takes an architecture, $\mathcal{A}_j$, and a service selection, Z_k, and produces an expected QoS value, $q_{i,j,k}$:

$$\mathbb{E}\left[q_{i,j,k}\right] = \mathcal{M}(\mathcal{A}_j, Z_k). \tag{4.2}$$

The global performance model is implemented using an expression tree that links together component performance models (see Section 2.5.6).

4.3.1 SSS Availability Model

This model computes the overall availability of the SSS. The expression trees used for SSS availability models have two layers: a root/parent node layer and a leaf/child node layer. The parent node of an availability model expression tree is a *multiplication product operation*, wherein all of the child nodes are muliplied together to provide the overall availability of the SSS. Each child node is a component performance model for availability. The component availability model dynamically links the appropriate architectural pattern availability model with the current observed availability of the SPs. There is one child node for each component that is a member of a given availability SSS.

4.3.2 SSS Throughput Model

This model determines the maximum possible throughput for a given SSS. The expression trees used for SSS throughput models also have just a root/parent node layer and a leaf/child

node layer. The parent node of a throughput model expression tree is a *minimum operation*, wherein the child with the lowest throughput value determines the overall throughput for the SSS. Each child node is a component performance model for throughput. The component throughput model dynamically links the appropriate child architectural pattern throughput model with the current observed capacity of the SPs. There is one child node for each component that is a member of a given throughput SSS.

4.3.3 SSS Security Option Model

This model determines the minimum level of security across the SSS. Similar to the previous two types of models, the expression trees used for SSS security option models have a root/parent node layer and a leaf/child node layer. The parent node of a security option model expression tree is a *minimum operation*, wherein the child with the lowest security level determines the overall security level for the SSS. Each child node contains the security level for the component it represents. There is one child node for each component that is a member of a given security option SSS.

4.3.4 SSS Execution Time Model

This model estimates the execution time for a given SSS. The expression trees used for SSS execution time models may have more than two layers, depending upon the structure of the SSS. The root/parent node of the execution time model expression tree is an *arithmetic summation operation*, wherein the child nodes evaulate themselves and the values they return to the parent are added together. The rest of the expression tree is constructed from *arithmetic summation operations*, *maximum operations*, and component performance models for execution time, according to the procedure shown in Algorithm 1. The inputs to the algorithm are $\mathcal{C}_i$, the set of components in SSS_i, and $\mathcal{L}_i$, the set of links connecting the components in the SSS (see Section 2.5.8 for more detail). The component performance model dynamically links the appropriate architectural pattern execution time model with the current observed execution time of the SPs.

4.4 Two-Level Optimization Search for Re-Architecting

As discussed in Section 4.2, the goal of the optimization procedure is to find an architecture, $\mathcal{A}^*$, and service selection, Z^*, that maximizes global utility, U_g. The U_g for a given

Algorithm 1 Generating Execution Time Performance Model for SSS_i

```
1: function BuildExeModelTree (C_i, L_i)
2: exprTree_i ← AllocateEmptyExpressionTree()
3: C_{i,1} ← GetFirstComponentOfSSS(C_i, L_i)
4: exprTree_i ← AddNodesToTree(C_{i,1}, C_i, L_i, exprTree_i)
5: return exprTree_i
6: end function

7: function AddNodesToTree (currentC, C_i, L_i, exprTree_i)
8: treeNodePointer ← InstantiateSummationOperationNode()
9: if IsEmpty(exprTree_i) then
10:   exprTree_i.root ← treeNodePointer
11: else
12:   LinkChildToParent(treeNodePointer, exprTree_i.cursor)
13: end if
14: exprTree_i.cursor ← treeNodePointer
15: flag ← true
16: while flag = true do
17:   type ← GetComponentType(currentC)
18:   if type = ACTIVITY then
19:     compModelPointer ← GetCompExeTimeModel(currentC)
20:     treeNodePointer ← InstantiateCompModelNode(compModelPointer)
21:     linkChildToParent(treeNodePointer, exprTree_i.cursor)
22:   else if type = FORK then
23:     treeNodePointer ← InstantiateMaximumOperationNode()
24:     LinkChildToParent(treeNodePointer, exprTree_i.cursor)
25:     exprTree_i.cursor ← treeNodePointer
26:     for all C_{i,j} such that LinkFromTo(currentC, C_{i,j}, L_i) = true do
27:       exprTree_i ← AddNodesToTree(C_{i,j}, C_i, L_i, exprTree_i)
28:       exprTree_i.cursor ← ParentOf(exprTree_i.cursor)
29:     end for
30:     currentC ← FindCorrespondingJoin(currentC, C_i, L_i)
31:     exprTree_i.cursor ← ParentOf(exprTree_i.cursor)
32:   else
33:     break while loop {currentC is a JOIN and need to return to the FORK for loop.}
34:   end if
35:   if NoOutGoingLink(currentC, L_i) = true then
36:     flag ← false
37:   else
38:     currentC ← FollowOutgoingLink(currentC, C_i, L_i)
39:   end if
40: end while
41: return exprTree_i
42: end function
```

architecture, $\mathcal{A}_i$, requires computation of the QoS metrics that are used to calculate the attribute utility functions. The models described in Section 4.3 cannot compute QoS metrics without the performance characteristics of the SPs in the service selection. In the ideal case, $\mathcal{A}_i$ is assessed using the service selection, $Z^*_{\mathcal{A}_i}$, that reaches the global maximum for U_g. Thus, for each evaluation of an architecture, $\mathcal{A}_i$, a search for a corresponding $Z^*_{\mathcal{A}_i}$ should be performed. This is accomplished by a two-level optimization search—architecture and service selection—shown in Fig. 4.1.

After the base architecture is generated, the SASSY user interface (Box 1) passes the base architecture to the architecture search module (Box 3). When either a new SASSY application is specified or when requirements on an existing SASSY application are modified, the SASSY user interface will send a re-architecting message to the architecture search module.

The performance monitor (Box 2) tracks the U_g of the SASSY appplication over time by monitoring the performance of the SSSes. If the U_g declines below a specified threshold, a re-architecting message is sent from the performance monitor to the architecture search module (Box 3).

The architecture search module looks for an architecture that will maximize U_g. To accomplish this goal, the architecture search module evaluates a series of proposed architectures using a heuristic search algorithm. A proposed architecture, $\mathcal{A}_i$, cannot be assessed without a corresponding optimized service selection, $Z^*_{\mathcal{A}_i}$. To acquire an optimized $Z^*_{\mathcal{A}_i}$, the architecture search module sends the proposed architecture to the service selection search module (Box 4).

The service selection search module attempts to find an optimized, $Z^*_{\mathcal{A}_i}$, for $\mathcal{A}_i$ given an evaluation budget constraint, N_Z. The service selection search module evaluates a series of proposed service selections. The service selection search module sends the proposed architecture, $\mathcal{A}_i$, and a proposed service selection, Z_j, to both the SSS performance modeler (Box 5) and the evaluation function (Box 6).

The SSS performance modeler applies the models discussed in Section 4.3 to $\mathcal{A}_i$ and Z_j and predicts QoS values for each defined SSS. The SSS performance modeler sends

the predicted QoS values to the evaluation function. The evaluation function applies the attribute utility functions to the QoS values and then computes the global utility function. The expected U_g is returned to the service selection module.

The service registry (Box 7) supports the service selection search by sending a list of the available SPs and their attributes to both the service selection search module and the SSS performance modeler.

The service selection module continues generating and evaluating proposed service selections until the evaluation search budget, N_Z, has been consumed. At that point, the service selection search module returns the best service selection found, $Z^*_{\mathcal{A}_i}$, to the architecture search module.

The architecture search module continues generating and evaluating architectures (with each evaluation requiring a service selection search). When either a time limit or an ar-

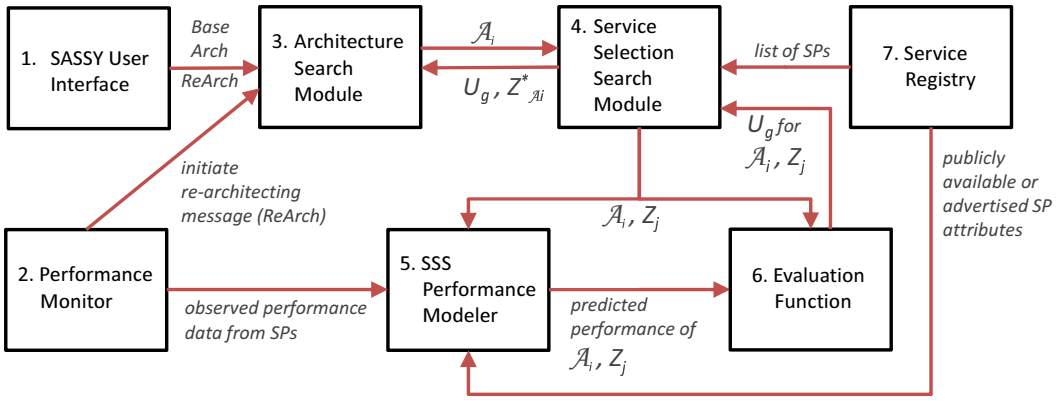


Figure 4.1: Basic framework for autonomic optimization in SASSY.

4.5 Heuristic Search Algorithms for Re-Architecting

The work in this section was originally presented in [39]. This section details the adaptation of the well-known heuristic algorithms described in Sections 2.3.4 and 2.3.5 to architecture search and service selection for the SASSY re-architecting problem. The four heuristic algorithms that have been adapted to SASSY are: 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 search algorithms 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.

4.5.1 Neighborhood Filtering (Hill-Climbing and Beam Search)

The rules for determining what constitutes a neighbor are 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.

To reduce the size of the neighborhood, a technique called a *neighborhood filter* was developed. A neighborhood 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. For each of the k SSSs, the j worst performing components for the given SSS metric are designated as 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 [94] that are expected to improve the metric of the SSS and 2) neighbors are produced by incrementing/decrementing the number of SPs 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 SP is identified according to the SSS metric. If an unused SP offers a performance improvement in the SSS metric, a neighbor is produced by substituting in the superior service instance.

4.5.2 Evolutionary Programming in SASSY

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 SPs in a composite component.
- Decreasing by one the number of SPs in a composite component.

The changes made in service selection mutation are substitutions of SPs. An adaptive step size is employed in service selection search. 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.

4.6 Heuristic Performance Differences

The work described in Section 3.2.3 demonstrated that autonomic controllers may experience significantly different levels of performance depending on the heuristic search algorithm employed. This section examines if the choice of heuristic search algorithms in SASSY is

also significant. The SASSY optimization problem is further complicated by the fact that it requires a two-layer search.

4.6.1 Impact of Architecture Search Algorithm

To investigate the impact of the architecture search algorithm, a limited experiment was conducted. In this experiment, two different heuristic pairs were used to solve the initial optimization problems generated by 100 simulations (using the simulation procedure described in Chapter 6). The two heuristic pairs used the same service selection search algorithm but used different architecture search algorithms.

The two heuristic pairs were generated using the meta-optimization process (presented in Section 4.7.2) on a 30 component SAS. Both heuristic pairs used the same evaluation budget: $N_{\mathcal{A}} = 100$ and $N_Z = 1,200$. The two heuristic pairs used the same heuristic algorithm for **service selection search**: opportunistic hill-climbing with no neighborhood filtering (`HC`). For the **architecture search**, the first heuristic pair, `HC-HC`, uses opportunistic hill-climbing with neighborhood filtering (SSS filter, $k = 5$, and component filter, $j = 2$). The second heuristic pair, `EP-HC`, uses evolutionary programming (`EP`) with non-overlapping populations, parent population size $M = 6$, offspring population size $K = 30$, and a step size of 2.0.

The architecture search trajectories for `HC-HC` and `EP-HC` are plotted against each other in Fig. 4.2, which depicts the average U_g as the search progresses. Over the first 30 evaluations, the difference in search performance is negligible. At that point, the U_g achieved by `HC-HC` plateaus, while `EP-HC` continues to find architectures with higher U_g. The 95% confidence intervals separate at about 70 evaluations. On these 100 inital optimization problems, `EP-HC` is clearly superior to `HC-HC`, demonstrating the significant impact of the architecture search algorithm.

4.6.2 Impact of Service Selection Search Algorithm

A second limited experiment was conducted to investigate the impact of the service selection search algorithm. In this experiment, a third heuristic pair was used to solve the same 100 initial optimization problems with the same evaluation budgets used by the previous two

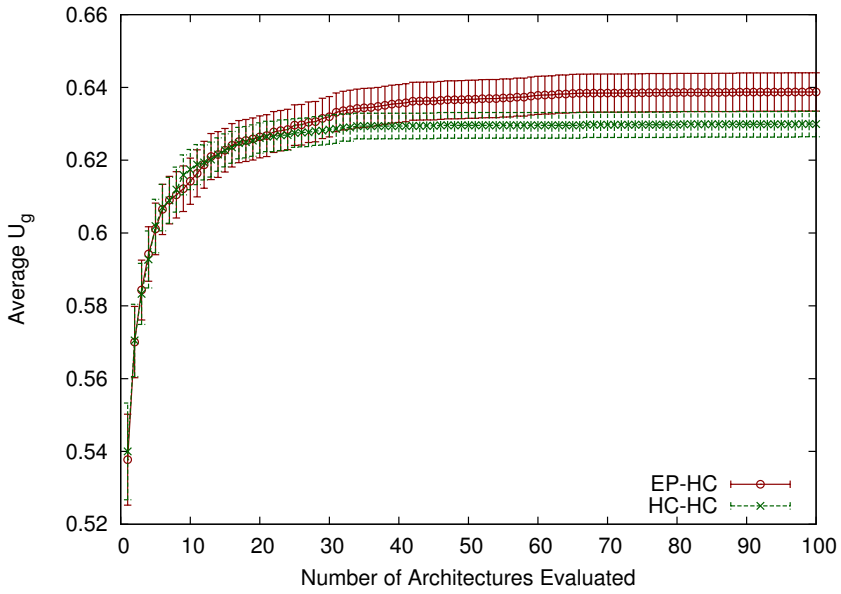


Figure 4.2: Average architecture search trajectory on 100 SAS-25 optimization problems with 95% CI bars.

heuristic pairs. This third heuristic pair, `HC-EP`, used the same **architecture search** algorithm as `HC-HC`. However `HC-EP` employed a different **service selection search** algorithm: evolutionary programming 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 service selection search trajectories for `HC-HC` and `HC-EP` are plotted against each other in Fig. 4.3. The difference in performance between the service selection heuristics is immediately clear—`HC-EP` far outperforms `HC-HC`. The difference between the heuristic pairs peaks at 280 service selection evaluations. After 1,200 service selection evaluations, `HC-HC` is closer to `HC-EP` but a significant difference remains. In essence, Fig. 4.3 shows that `HC-EP` converges on better solutions more rapidly than `HC-HC`.

The U_g values returned by the service selection search module to the architecture search module are used to evaluate architectures. As such, these values also inform decisions about which portions of the architecture space to invest in future search resources. Reductions in service selection search performance may also reduce the signal-to-noise ratio in the data used to guide the architecture search.

The next step in this analysis was to examine how the difference in service selection

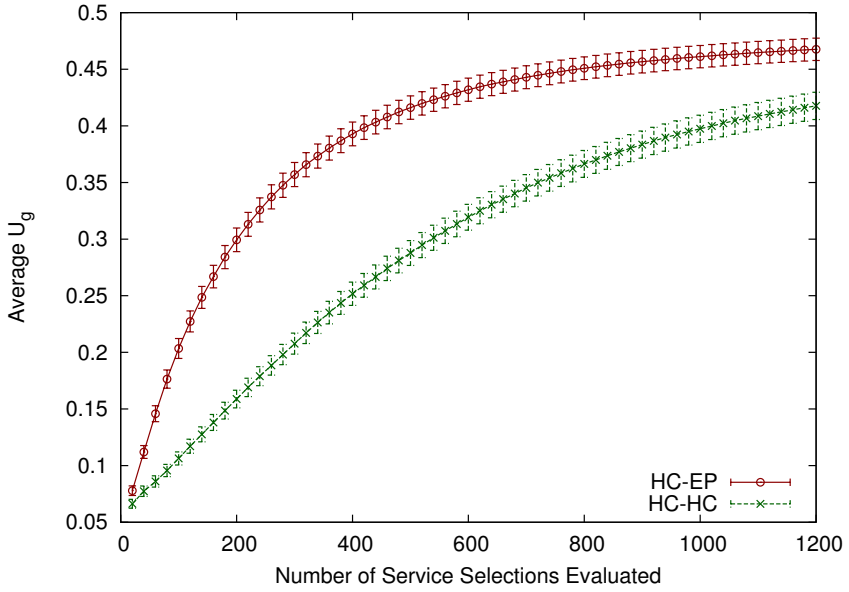


Figure 4.3: Average service selection search trajectory on 100 SAS-25 optimization problems with 95% CI bars.

search performance affected the architecture search. The architecture search trajectories for `HC-HC` and `HC-EP` are plotted against each other in Fig. 4.4. The curves of the trajectories in Fig. 4.4 have approximately the same shape, but the best U_g found by `HC-HC` trail those found by `HC-EP` by a fairly constant margin.

These limited experiments confirm that the choice of heuristic algorithms for both architecture search and service selection search does impact search performance.

4.7 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, this section assumes that all self-optimizing systems can be gauged with a global utility function. This work was originally reported in [38].

Formally, self-optimization can be specified as:

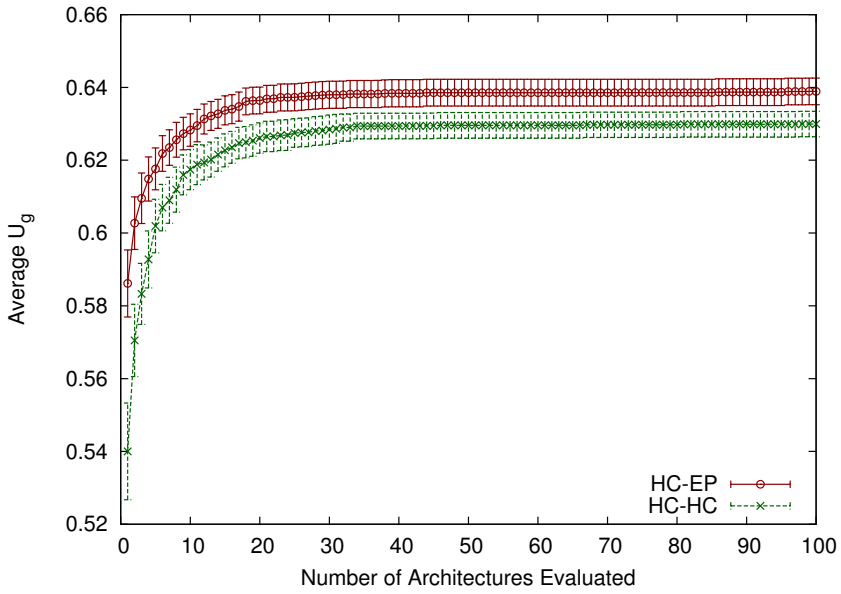


Figure 4.4: Average architecture search trajectory on 100 SAS-25 optimization problems 95% CI error bars.

Find a system state S^* such that

$$S^* = \text{argmax}_S \; U_g(S, \mathcal{K}) \tag{4.3}$$

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 4.4 shows the optimization process, B, producing an approximately optimized state, S_a^* with optimization algorithm, $\mathcal{H}$.

$$S_a^* = B(\mathcal{H}, \mathcal{K}). \tag{4.4}$$

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:

$$\overline{U}_{\mathcal{H}} = \mathbb{E}\ [U_g(S_a^*)] = \mathbb{E}\ [U_g(B(\mathcal{H}, \mathcal{K}))]. \tag{4.5}$$

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

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

$$\mathcal{H}^* = \text{argmax}_{\mathcal{H}}\ \mathbb{E}\ [U_g(B(\mathcal{H}, \mathcal{K}))] \tag{4.6}$$

$$t_{\mathcal{H}} \leq t_L \tag{4.7}$$

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

4.7.1 Meta-Optimization in SASSY

There are two NP-hard optimization problems that need to be solved in near real-time for SASSY [92]:

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

$$(\mathcal{A}^*, Z^*) = \text{argmax}_{(\mathcal{A},Z)}\ U_g(\mathcal{A}, Z, \mathcal{K}). \tag{4.8}$$

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 [92].

The optimization process, B, in SASSY requires two algorithms: $\mathcal{H}_\mathcal{A}$ for the architecture search and $\mathcal{H}_Z$ for the service selection search. Equation 4.9 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 $\mathcal{A}_a^*$.

$$(\mathcal{A}_a^*, Z_a^*) = B(\mathcal{H}_\mathcal{A}, \mathcal{H}_Z, \mathcal{A}_c, \mathcal{K}). \tag{4.9}$$

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

$$\overline{U}_{\mathcal{H}_\mathcal{A},\mathcal{H}_Z} = \mathbb{E}\left[U_g(\mathcal{A}_a^*, Z_a^*)\right] = \mathbb{E}\left[U_g(B(\mathcal{H}_\mathcal{A}, \mathcal{H}_Z, \mathcal{A}_c, \mathcal{K}))\right]. \tag{4.10}$$

Equation 4.11 describes the meta-optimization problem in SASSY:

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

$$(\mathcal{H}_\mathcal{A}^*, \mathcal{H}_Z^*) = \text{argmax}_{(\mathcal{H}_\mathcal{A},\mathcal{H}_Z)}\ \mathbb{E}\left[U_g(B(\mathcal{H}_\mathcal{A}, \mathcal{H}_Z, \mathcal{A}_c, \mathcal{K}))\right] \tag{4.11}$$

$$t_{(\mathcal{H}_\mathcal{A},\mathcal{H}_Z)} \leq t_L \tag{4.12}$$

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 (as evidenced in Section 4.6). Each of these heuristic search algorithms require multiple parameter settings that can have potentially large impacts on the optimization process performance.

4.7.2 Meta-Optimization Framework

As demonstrated in Equations 4.6 and 4.11, certain inputs are required in the meta-optimization process. In the general case, the operating environment state, $\mathcal{K}$, is required 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, the following three-step meta-optimization process is proposed:

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.

Generating Candidate Problems in SASSY

To capture a candidate sample problem set in SASSY, the SASSY system operates 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. For a more detailed explanation of the simulation used, see Section 6.2. 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 Fig. 6.5, between three and ten SPs were randomly generated for each of the 65 activities, yielding an overall total of 404 SPs. A relatively long initial optimization search found an optimized starting architecture, $\mathcal{A}_i$, and an optimized SP selection, Z_i. The instantiated SASSY system operated using the beam search/evolutionary programming `BS-EP` heuristic search algorithm pair (this heuristic pair was originally from a meta-optimization process on a 30 component SAS). The SASSY

system, starting with $\mathcal{A}_i$ and Z_i, was exposed to simulated SP failures, simulated SP degradation, and simulated SP repair events over time. The SASSY system went through 26 such simulations and captured 1% of the encountered optimization problems encountered by the SASSY autonomic controller. This process generated 1,041 candidate sample problems.

Selecting Finalist Problems in SASSY

Chapter 5 will demonstrate 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 (see Chapter 5). To improve the probability of including one or more challenge problems in the finalist subset, diversity is prioritized when choosing finalists from candidates.

To develop a diverse finalist subset, two summary statistics are considered:

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

To pick a diverse group of finalist problems, the selected finalist problems are 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 Fig. 4.5 and are labeled A through L.

Applying Meta-Optimization Procedure

Figure 4.6 describes the meta-optimization procedure 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

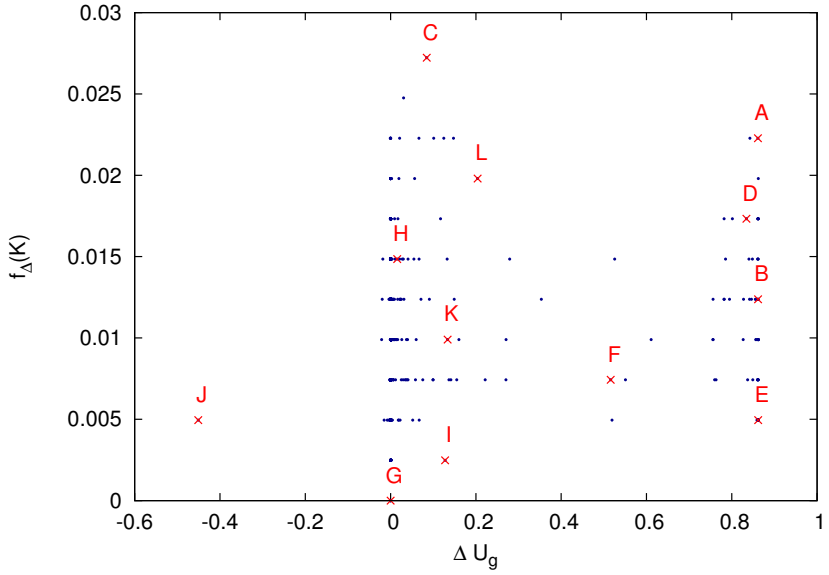


Figure 4.5: The candidate problem set plotted using summary statistics. The twelve finalist problems are labeled A-L and marked with red x's.

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 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

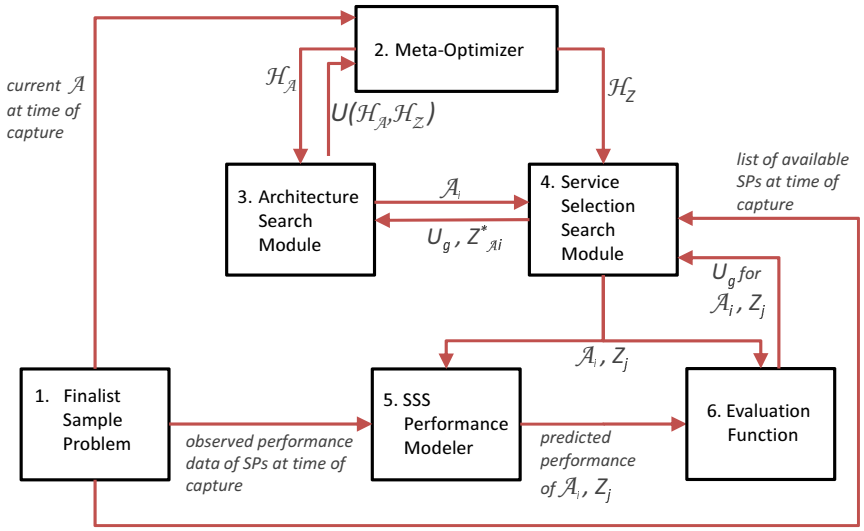


Figure 4.6: The meta-optimization procedure applied to SASSY.

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.

Genetic Algorithm for the Meta-Optimizer

A genetic algorithm is employed as the 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 4.1 and Table 4.2. The

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
# SSSes in filter	hill-climbing & beam search	integer	1	13	1	4
# 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	0.04	0.7	4
final probability	simulated annealing	floating point	0.00001	0.00016	0.00001	4

Table 4.1: Composition of architecture search binary string.

genotype of the heuristic search algorithm pair is formed by concatenating these two binary strings.

The service selection search budget parameter, N_Z, in Table 4.2 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:

$$N_{\mathcal{M}} = N_{\mathcal{A}} \times N_Z \tag{4.13}$$

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.

As in most genetic algorithms, the size of the parent population and offspring population are equal (in this work the population size is set to 15). Parent selection is conducted with the probabilistic linear rank method described in Section 2.3.5 and Equation 2.15.

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

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
# SSSes in filter	hill-climbing & beam search	integer	1	13	1	4
# 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	0.04	0.7	4
final probability	simulated annealing	floating point	0.00001	0.00016	0.00001	4

Table 4.2: Composition of service selection search binary string.

transcription [24].

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 [113] 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.

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, $U_{\mathcal{H}_\mathcal{A},\mathcal{H}_Z}$, is computed as follows:

$$U_{\mathcal{H}_\mathcal{A},\mathcal{H}_Z} = \frac{\sum_{i=1}^{n} U_g(\mathcal{A}_i, Z_i)}{n} \tag{4.14}$$

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.

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

Table 4.3: Resulting heuristic pairs for representative problems A through D.

Parameter	problem E	problem F	problem G	problem H
arch. search budget, $N_{\mathcal{A}}$	19	20	22	20
arch. search alg.	hill-climbing	beam search	hill-climbing	sim. annealing
arch. search mode	greedy	no LL req.	opportunistic	N/A
arch. # of filter SSSes	3	4	11	N/A
arch. # of filter comp.	1	1	3	N/A
arch. beam width	N/A	5	N/A	N/A
arch. ini. prob.	N/A	N/A	N/A	0.26
arch. final prob.	N/A	N/A	N/A	0.0008
serv. sel. search budget, N_Z	2,475	2,275	2,100	2,375
serv. sel. search alg.	evo. prog.	evol. prog.	evol. prog.	evol. prog.
serv. sel. par. pop. size	1	4	1	3
serv. sel. off. pop. size	6	4	4	18
serv. sel. overlap pop.	false	true	true	true
serv. sel. ini. step size	2.5	4.5	4.5	2.5
serv. sel. adapt. step fact.	1.5	1.5	1.0	1.0

Table 4.4: Resulting heuristic pairs for representative problems E through H.

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 $U_{\mathcal{H}_{\mathcal{A}},\mathcal{H}_Z}$ 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 Tables 4.3, 4.4, and 4.5.

From the results in Tables 4.3, 4.4, and 4.5, 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

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

Table 4.5: Resulting heuristic pairs for representative problems I through L.

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.

Figures 4.7 and 4.8 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.

Each of the finalist sample problems has a different y-scale. To gauge the overall convergence of the meta-optimization genetic algorithm, the search performance is normalized against the best U_g found during the entire meta-optimization search. A plot of the normalized convergence can be found in Fig. 4.9.

4.8 Concluding Remarks Regarding Architecture Search

This chapter has described in detail the SASSY architecture optimization problem and the heuristic search algorithms that can be used to solve it. Small experiments have shown that the choice of heuristic search algorithm matters. A meta-optimization technique was presented to facilitate finding good heuristic search algorithms. The next chapter considers how to fully leverage the heuristic search algorithms produced by the meta-optimization process.

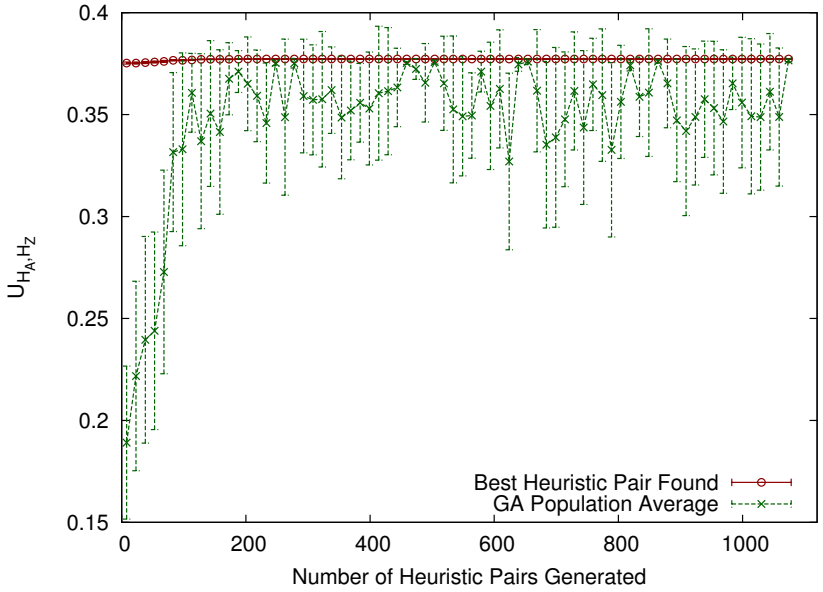


Figure 4.7: Heuristic pair performance on problem D with 95% CI bars.

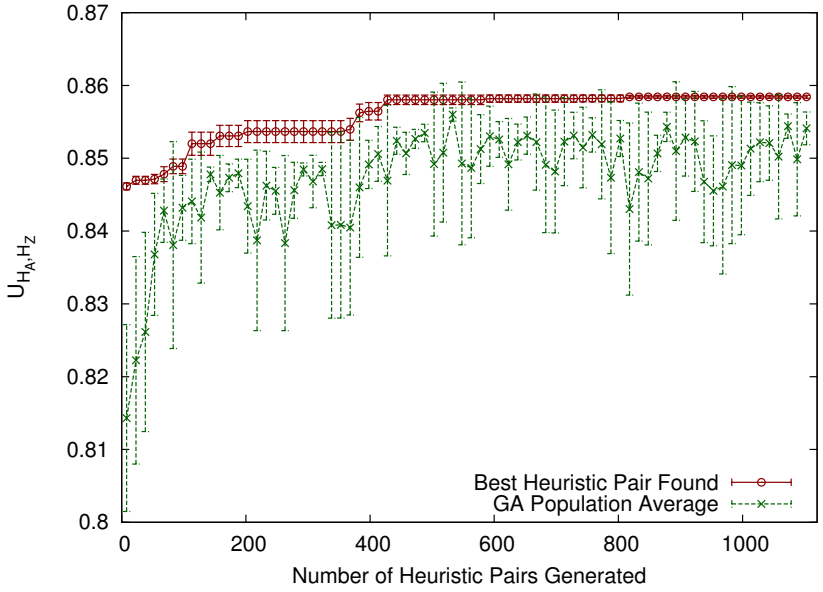


Figure 4.8: Heuristic pair performance on problem F with 95% CI bars.

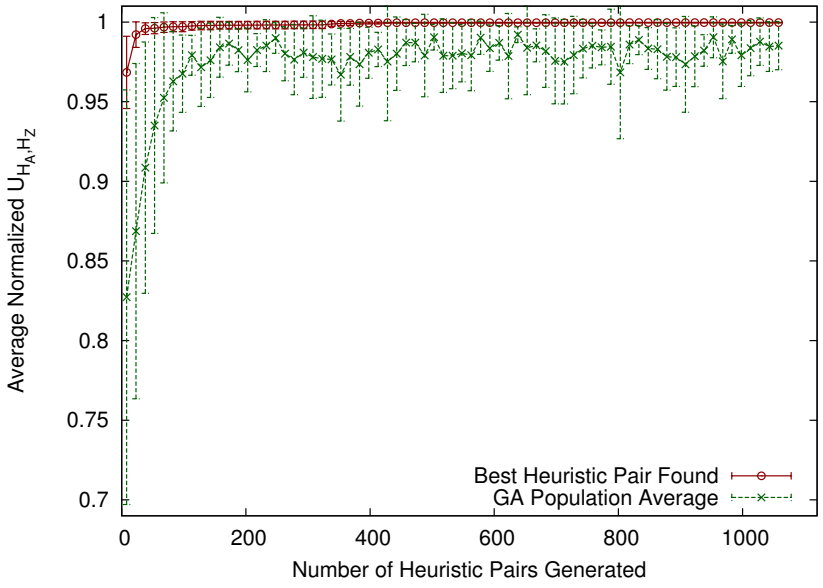


Figure 4.9: Normalized heuristic pair performance across all problems with 95% CI bars.

Chapter 5: Meta-Controller Approaches

This chapter describes four distinct meta-controller methods for autonomic optimization in SASSY. In this context, a meta-controller seeks to dynamically improve the performance of the autonomic controller. Meta-controllers in SASSY observe the performance of heuristic search algorithms and use these observations to determine which heuristic search algorithms should be used when the need for architecture optimization arises.

The different meta-controller approaches each employ machine learning but the degree of sophistication ranges from the simple (`Overall Best`) to the complex (SVM-based `Context Best` meta-controllers). By applying machine learning to the operation of the autonomic controller, meta-controllers seek to increase the degree of autonomicity from autonomous to autonomic (see Section 1.2.3 for more information on the degrees of autonomicity).

The first section of this chapter describes the framework common to all of the meta-controller approaches. The `Overall Best` meta-controller is described in the second section. The third section describes the three `Context Best` meta-controller methods (KNN, offline SVM, and online SVM). The final section of the chapter provides concluding remarks regarding meta-controllers.

5.1 Framework for the Meta-Controller

This section describes the framework for self-adaptation and architecture/component selection optimization. Figure 5.1 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.

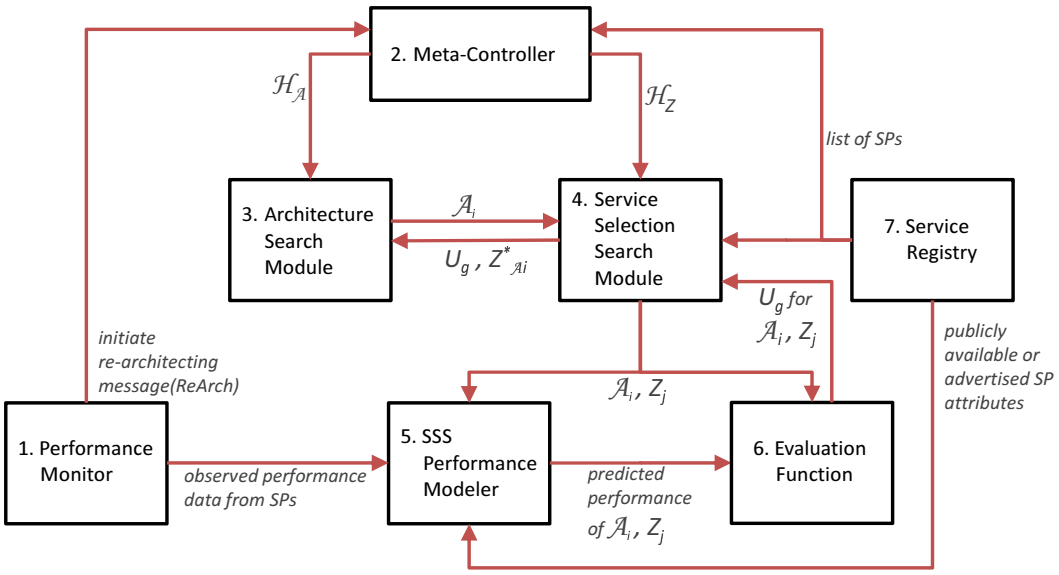


Figure 5.1: Data flows in the meta-controller monitoring and optimization framework.

In the first case, to initiate an architecture optimization search, the performance monitor sends a message to the meta-controller (box 2). The meta-controller selects an appropriate heuristic search procedure, $\mathcal{H}_{\mathcal{A}}$ for the architecture search module (box 3) and a potentially different heuristic search procedure, $\mathcal{H}_{Z}$, for the service selection search module (box 4).

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

With the completion of a service selection search instance, $\mathcal{H}_\mathcal{A}$ uses $U_g(\mathcal{A}_i, Z^*_{\mathcal{A}_i})$ as the fitness score for $\mathcal{A}_i$ and continues the search. The heuristic $\mathcal{H}_\mathcal{A}$ persists searching until some exit criterion is met (e.g., threshold utility is achieved or evaluation budget consumed). When $\mathcal{H}_\mathcal{A}$ completes, the architecture search module (box 3) sends $\mathcal{A}^*$ and Z^* to the change planner/manager (not shown in Fig. 5.1). The change planner/manager then executes a plan for online evolution or adaptation of the running system.

This framework assumes the existence of a service registry (box 7) that includes QoS levels of the SPs listed in the registry [20]. This information is required by three modules: the meta-controller (box 2) that uses this information when selecting $\mathcal{H}_\mathcal{A}$ and $\mathcal{H}_Z$, 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 optimization search as described Chapter 4. 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.

Each time that 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 architecture optimization search completes, the meta-controller stores a *result tuple* of $\mathcal{P}$: ($\mathcal{H}_\mathcal{A}$, $\mathcal{H}_Z$, U_g^{best}). After the re-architecting process completes, the meta-controller begins a

training process testing other heuristic search pairs from the candidate list against $\mathcal{P}$ and storing the outcome in the result tuple. This training process can be preempted by new re-architectecting searches and resumed later.

5.2 Overall Best Heuristic Pair

The `Overall Best` meta-controller attempts to determine the overall best candidate heuristic pair, $(\mathcal{H}^*_{\mathcal{A}}, \mathcal{H}^*_{Z})$, over the entire range of re-architecting optimization problems encountered by a SASSY application (see Chapter 4 for more information about heuristic pairs). Each time a result tuple is stored, the `Overall Best` meta-controller updates the average U_g^{best} for the given heuristic pair. When it is time for the `Overall Best` meta-controller to make a decision, it chooses the heuristic pair that has produced the highest average U_g^{best}.

5.3 Context Best Heuristic Pair

It is unlikely that there is a single heuristic search pair that outperforms all other heuristic search pairs over the potential optimization problem space. A certain heuristic pair may dominate a portion of the optimization problem space, while other heuristic pairs may dominate other portions of the space. A `Context Best` meta-controller attempts to determine the overall best candidate heuristic pair given specific features of $\mathcal{P}$.

In many architecture search problems, a reasonable local-optimum architecture is near the starting architecture. When encountering such problems, the autonomic controller is best served by using exploitative 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. When solving these problems, the autonomic controller is best served using exploratory 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 event can impact the expected distance of near-optimal architectures from the starting architecture. Thus, measurements of service environment changes, such as SP availability,

may offer insight into the likelihood of proximate local-optimum 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, such as a reduction in throughput in an SSS, 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 pairs.

5.3.1 Characterizing the Optimization Problem

An accurate and relevant representation of the optimization problem, $\mathcal{P}$, is required for a machine learning approach to successfully train. The representation used for the `Context Best` meta-controller is shown in Table 5.1. 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 (see Section 2.5.5 for more detail on architectural patterns); one and only one of these three features must be set to true for each component. The current level feature 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.

5.3.2 Processing the Training Set

The `Context Best` meta-controller conducts its training process by testing all candidate heuristic pairs against the features of the problem, $\mathcal{F}(\mathcal{P})$, until 50 result tuples, $[\mathcal{P}, \mathcal{H}_{\mathcal{A}}, \mathcal{H}_{Z}, U_g^{\text{best}}]$,

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 5.1: Features of the machine learning problem, $\mathcal{F}(\mathcal{P})$.

have been generated for each candidate heuristic pair on a given problem. Whenever a result tuple is stored, a `Context Best` meta-controller extracts the $\mathcal{F}(\mathcal{P})$ in Table 5.1. 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, see Appendix A for more details). If the training set already contains a matching training record, the new results are appended to the pre-existing record in the training set.

Machine learning algorithms that make calculations in Euclidean space are sensitive to differences in the scale of the ranges: features with relatively large ranges will be emphasized to the detriment of features with relatively small ranges. If the information contained in features with relatively small data ranges is significant to the classification problem, the performance of the machine learning algorithm will suffer [17]. To address this issue, the `Context Best` meta-controller tracks maximum and minimum values for each feature in the training set. These maximum and minimum values are used to maintain a copy of the training set where each feature is normalized to the range of $[-1 : 1]$. The maximum and minimum values are also used to scale the features of encountered problems not yet in the training set.

5.3.3 Decision Making

When the `Context Best` meta-controller needs to select a candidate heuristic pair, 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 pair has the best recorded performance in that training record.

If no matching training record is found, the `Context Best` meta-controller normalizes $\mathcal{F}(\mathcal{P}_{current})$ using the maximum and minimum values from the training set. Next the `Context Best` meta-controller employs a machine learning algorithm to select the expected best heuristic pair for $\mathcal{F}(\mathcal{P}_{current})$.

5.3.4 KNN Meta-Controller

The first machine learning technique that was applied to the `Context Best` meta-controller is the k-nearest neighbor (KNN) algorithm that was described in Section 2.4.1. The training set serves directly as the classification model in KNN—no training is required. Using KNN, the meta-controller chooses a heuristic pair 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 problem.

4. If one heuristic pair received more votes than any other, select that heuristic pair. If there is a tie in the voting, select the heuristic pair from the training record closest to $\mathcal{F}(\mathcal{P}_{current})$.

For moderate to large SASSY systems, $\mathcal{F}(\mathcal{P}_{current})$ will have hundreds of features. With a large number of features, the computation of the Euclidean distance may not be trivial on large training sets, especially given the near real-time requirements for the SASSY meta-controller.

5.3.5 Offline Training SVM Meta-Controller

As discussed in Section 2.4.2, support vector machines (SVM) with kernel methods offer a sophisticated alternative to KNN. From the perspective of an autonomic meta-controller, SVMs have some attractive properties. The process of selecting a heuristic pair requires just two steps: 1) project $\mathcal{F}(\mathcal{P}_{current})$ to a higher dimensional space using the selected kernel method and 2) compute a vector multiplication to determine on which side of the hyperplane decision boundary $\mathcal{F}(\mathcal{P}_{current})$ lies. On modern computing systems, these calculations are trivial.

SVM does require a significant amount of effort for training. In particular, SVM is sensitive to the selection of kernel parameters and C, which controls the size of the margin around the decision boundary. To maximize the SVM's prediction accuracy, it is necessary to optimize these parameters. The accuracy for a given set of SVM parameters can be assessed through n-fold cross-validation (see Section 2.4.2). Thus, searching for optimal parameters requires significant effort. Training the SVM and finding optimal kernel parameters can either be performed offline or performed online between optimization searches when the hardware of the autonomic controller would otherwise be idle.

Traditional SVM supports two-class pattern recognition. Extensions for multi-class SVM are available but may take longer to train and require proportionally larger data sets [55]. Given the limited training sets that will be available to the SASSY meta-controller, the `Offline SVM` meta-controller uses two-class pattern recognition. This means that two different heuristic pairs must be selected from the candidate pairs; for example on the SAS-65, an `Offline SVM` meta-controller might select heuristic pairs E and G from Table 4.4 in Section 4.7.2. Problems where E outperforms G are labeled E, and vice versa; the labeling process is explained in Equation 5.7. These labels are mapped into the two classes of the SVM.

An `Offline SVM` meta-controller employing an offline training set requires the completion of the following actions to build an SVM model.

1. Collect and study a set of training problems.

2. Find the most suitable heuristic pair combination.

3. Remove noisy training problems from the training set.

4. Determine penalty weights for mis-classification.

5. Optimize SVM parameters.

6. Train the SVM model.

Collect and Study Training Problems

Similar to the collection of candidate problems for the meta-optimizer described in Section 4.7.2, problems for the offline SVM training set are collected while the SASSY system executes with a simple autonomic controller (i.e., without using a meta-controller) in a simulated service environment (for more details see Section 6.2). This problem collection process is different from that used by the `Overall Best` and `Context Best` KNN meta-controllers, which build training sets by collecting problems as they are encountered. The problems for the offline SVM training set are collected through random sampling—this helps to avoid oversampling small portions of the problem space. As each training problem $\mathcal{P}_t$, is collected, $\mathcal{F}(\mathcal{P}_t)$ is computed and stored with $\mathcal{P}_t$.

As the SASSY simple autonomic controller executes in the simulated environment, it is important that the controller use one of the best performing heuristic search algorithm pairs because experience has shown that the SVM is sensitive to the range of U_g values in the feature space. The training set problems collected by using an inferior heuristic pair are likely to have a different range of U_g values than those encountered by a meta-controlled SASSY system—this difference in U_g range increases the odds that the trained meta-controller will later choose the wrong heuristic pair on encountered optimization problems.

After collecting a sufficient number of training set optimization problems, it is necessary to investigate how the candidate heuristic pairs—described in Chapter 4—perform on them. Each candidate heuristic pair is tested against each training set problem 50 times. The results of this testing are stored in the result tuples described in Section 5.1.

Find Most Suitable Heuristic Pair Combination

The step of finding the most suitable heuristic pair combination consists of four sub-steps designed to maximize the SVM meta-controller's performance by considering three key factors. For each possible combination of two candidate heuristic pairs, the system

1. assesses potential performance,
2. assesses the balance of the training set,
3. assesses complementary behavior, and
4. makes an aggregate assessment.

The subsequent sections provide a detailed explanation of each sub-step.

Find Suitable Pairs: Assess Potential Performance Sub-Step

The purpose of the first sub-step is to determine the expected performance of a heuristic pair combination, $\mathbb{E}\ [U_g^{best}(\mathtt{mc}_{\mathtt{perfect}}((\mathcal{H}_{\mathcal{A}}^j, \mathcal{H}_Z^j), (\mathcal{H}_{\mathcal{A}}^k, \mathcal{H}_Z^k))]$ where $\mathtt{mc}_{\mathtt{perfect}}$ represents a meta-controller that always predicts the optimal heuristic pair (either j or k) for each encountered problem.

For each problem, $\mathcal{P}_t$, in the training set, the system computes $\overline{U}_{\mathcal{P}_t,\mathcal{H}_{\mathcal{A}}^j,\mathcal{H}_Z^j}$ from the result tuples as follows:

$$\overline{U}_{\mathcal{P}_t,\mathcal{H}_{\mathcal{A}}^j,\mathcal{H}_Z^j} = \frac{1}{n}\sum_{i=1}^{n} U_g^{\text{best}}(\mathcal{P}_t, \mathcal{H}_{\mathcal{A}}^j, \mathcal{H}_Z^j, i) \tag{5.1}$$

where n is the number of result tuples that match the combination $[\mathcal{P}_t, \mathcal{H}_{\mathcal{A}}^j, \mathcal{H}_Z^j]$, and i is the index of the result tuple amongst all other matches of a particular $[\mathcal{P}_t, \mathcal{H}_{\mathcal{A}}^j, \mathcal{H}_Z^j]$. Now, $\mathbb{E}\ [U_g^{best}(\mathtt{mc}_{\mathtt{perfect}}((\mathcal{H}_{\mathcal{A}}^j, \mathcal{H}_Z^j), (\mathcal{H}_{\mathcal{A}}^k, \mathcal{H}_Z^k))]$ can be computed:

$$\mathbb{E}\ [U_g^{best}(\mathtt{mc}_{\mathtt{perfect}}((\mathcal{H}_{\mathcal{A}}^j, \mathcal{H}_Z^j), (\mathcal{H}_{\mathcal{A}}^k, \mathcal{H}_Z^k)))] \ = \ \frac{1}{N}\sum_{t=1}^{N} \max(\overline{U}_{\mathcal{P}_t,\mathcal{H}_{\mathcal{A}}^j,\mathcal{H}_Z^j}, \overline{U}_{\mathcal{P}_t,\mathcal{H}_{\mathcal{A}}^k,\mathcal{H}_Z^k}) \tag{5.2}$$

where N is the number of problems in the training set.

For the purpose of providing a benchmark, against which the performance of a theoretically perfect meta-controller can be measured, the SASSY meta-controller framework computes the best performance that a simple autonomic controller could provide. The best heuristic pair for the overall training set, $(\mathcal{H}^*_{\mathcal{A}}, \mathcal{H}^*_Z)$, is determined as follows:

$$(\mathcal{H}^*_{\mathcal{A}}, \mathcal{H}^*_Z) \;=\; \text{argmax}_{(\mathcal{H}^i_{\mathcal{A}}, \mathcal{H}^i_Z)}\; \mathbb{E}\left[U_g^{best}(\mathcal{H}^i_{\mathcal{A}}, \mathcal{H}^i_Z)\right] \tag{5.3}$$

$$\mathbb{E}\left[U_g^{best}(\mathcal{H}^i_{\mathcal{A}}, \mathcal{H}^i_Z)\right] \;=\; \frac{1}{N}\sum_{t=1}^{N} \overline{U}_{\mathcal{P}_t, \mathcal{H}^i_{\mathcal{A}}, \mathcal{H}^i_Z}. \tag{5.4}$$

For the purposes of the aggregate assessment, a relative performance metric, `PotentialGain`, for each heuristic pair combination, $(\mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z), (\mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z)$ is computed.

$$\begin{aligned}\texttt{PotentialGain}((\mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z), (\mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z)) \;=\;& \mathbb{E}\left[U_g^{best}(\texttt{mc}_{\texttt{perfect}}((\mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z), (\mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z))\right] \\ &- \mathbb{E}\left[U_g^{best}(\mathcal{H}^*_{\mathcal{A}}, \mathcal{H}^*_Z)\right]\end{aligned} \tag{5.5}$$

The relative performance metric is converted into a score, `PotentialGainScore`, that is used to compute the aggregate assessment.

$$\begin{aligned}\texttt{PotentialGainScore}((\mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z), (\mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z)) \;=\; \max(0, &\texttt{PotentialGain}((\mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z), \\ &(\mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z)))\end{aligned} \tag{5.6}$$

The `PotentialGainScore` is one component of the `AggregateScore` that will be used to determine the best combination of heuristic pairs.

Find Suitable Pairs: Assess Balance Sub-Step

Properly training an SVM is challenging when one class has significantly more sample problems than the other class in the training set. Such a training set is termed unbalanced [17].

A vanilla SVM training on an unbalanced training set will tend to favor the class more heavily represented. This favoritism can become extreme with small, heavily unbalanced training sets resulting in a biased decision rule produced by the SVM. The labeling process for heuristic pair combinations will often produce unbalanced training sets. Some techniques including boosting [109] and penalty weights (which will be applied later) can be used to counter this favoritism and resulting bias. These techniques improve the performance, but when possible it is best to use more balanced training sets.

The first step in evaluating the balance of the training set produced for a given heuristic pair combination is to label the sample problems in the training set with the heuristic pair expected to provide the best performance for each. Each problem, P_t, in the training set is labeled with either heuristic pair j or heuristic pair k as follows:

$$\texttt{label}(\mathcal{P}_t) \;=\; \text{argmax}_{i\,\in\{j,k\}}\; \overline{U}_{\mathcal{P}_t,\mathcal{H}^i_{\mathcal{A}},\mathcal{H}^i_Z}. \tag{5.7}$$

A metric, `Balance`, that measures the degree of balance can be computed as follows:

$$\texttt{Balance}(\mathcal{H}^j_{\mathcal{A}},\mathcal{H}^j_Z),(\mathcal{H}^k_{\mathcal{A}},\mathcal{H}^k_Z)) \;=\; \frac{2}{N}\times\min(n_j,n_k) \tag{5.8}$$

where n_j is the number of problems labeled with heuristic pair j, and n_k is the number of problems labeled with heuristic pair k. The total number of problems in the training set is N.

The `Balance` metric can have values in the range $[0,1]$. A perfectly balanced training set yields a value of one. As the training set becomes more unbalanced, the `Balance` metric declines. Extremely unbalanced training sets are especially harmful with the small training sets used by the SASSY meta-controller, so a further penalty is required to discourage the more extremely unbalanced training sets. The `BalanceScore` does this by taking the square of the `Balance` metric.

$$\texttt{BalanceScore}((\mathcal{H}^j_{\mathcal{A}},\mathcal{H}^j_Z),(\mathcal{H}^k_{\mathcal{A}},\mathcal{H}^k_Z)) \;=\; (\texttt{Balance}((\mathcal{H}^j_{\mathcal{A}},\mathcal{H}^j_Z),(\mathcal{H}^k_{\mathcal{A}},\mathcal{H}^k_Z)))^2 \tag{5.9}$$

The `BalanceScore` is another component of the `Aggregate Score` that will be used to

determine the best combination of heuristic pairs.

Find Suitable Pairs: Assess Complementary Behavior Sub-Step

Correctly determining the best heuristic pair for a given problem depends upon successfully identifying a relationship between $\mathcal{F}(\mathcal{P}_t)$ and $\texttt{label}(\mathcal{P}_t)$. For heuristic pairs with similar search behaviors, this relationship may be difficult to determine. For these heuristic pairs, the labeling process may be dominated by noise or esoteric phenomena not captured in $\mathcal{F}(\mathcal{P}_t)$.

To improve its chances of building a successful SVM model, the meta-controller should train with a combination of heuristics that behave differently. The behavior of a heuristic pair on a given optimization problem is driven by the interaction of the algorithm's logic and the topology of U_g over the architecture and service selection spaces. These interactions can be complex and difficult to predict.

The training set contains useful empirical data for analyzing behavioral differences between the heuristic search pairs. This behavioral analysis begins by computing the average performance of the candidate heuristics for each problem as follows:

$$\overline{U}_{\mathcal{P}_t} \;=\; \frac{1}{n_c}\sum_{i=1}^{n_c} \overline{U}_{\mathcal{P}_t,\mathcal{H}^i_{\mathcal{A}},\mathcal{H}^i_Z} \tag{5.10}$$

where i is the index of a candidate heuristic pair and n_c is the number of candidate heuristic pairs being considered.

Using $\overline{U}_{\mathcal{P}_t}$ as a basis, the relative performance, `RelPerf`, of a heuristic pair on a given problem can be computed.

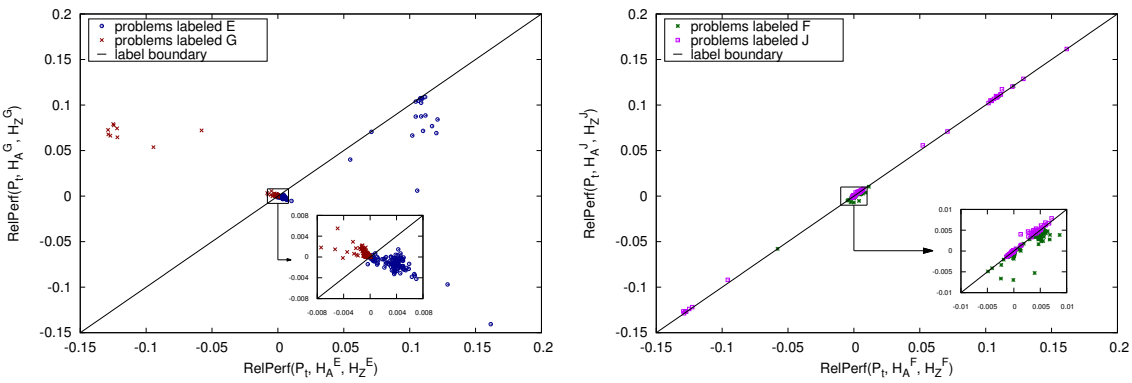
$$\texttt{RelPerf}(\mathcal{P}_t, \mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z) \;=\; \overline{U}_{\mathcal{P}_t,\mathcal{H}^j_{\mathcal{A}},\mathcal{H}^j_Z} \;-\; \overline{U}_{\mathcal{P}_t}. \tag{5.11}$$

$$\texttt{RelPerf}(\mathcal{P}_t, \mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z) \;=\; \overline{U}_{\mathcal{P}_t,\mathcal{H}^k_{\mathcal{A}},\mathcal{H}^k_Z} \;-\; \overline{U}_{\mathcal{P}_t}. \tag{5.12}$$

The `RelPerf` metric provides indications of how heuristic pair behavior is impacting performance. Figure 5.2 plots training set problems by the `RelPerf` metrics of two different

heuristic pair combinations. The heuristic pairs shown in Fig. 5.2 were developed by the meta-optimizer described in Section 4.7 and the parameters for the heuristic pairs can be found in Tables 4.4 and 4.5.

When the labels match the heuristic pairs used to produce `RelPerf` metrics, the diagonal line $x = y$ forms a labeling boundary. In Fig. 5.2b, all of the plotted training set problems are close to the labeling boundary. In Fig. 5.2a, there are clusters of training set problems some distance away from the label boundary. Also of interest is the shape of the main cluster of problems shown in the zoomed-in plots. In Fig. 5.2b, the main cluster is narrow and follows the label boundary. The main cluster in Fig. 5.2a is roughly the shape of a bow tie with an orientation orthogonal to the labeling boundary. The distribution of these points are an indication that the behaviors of E and G are more complementary than the behaviors of F and G.



(a) E and G plotted with `RelPerf` of E and G. (b) F and J plotted with `RelPerf` of F and J.

Figure 5.2: Figures (a) and (b) show two possible heuristic pair combinations for a training set. A label for the training set problem, $\mathcal{P}_t$ is determined by which heuristic pair yields a superior value for `RelPerf`. The heuristic pairs used to generate the `RelPerf` values are the same heuristic pairs used for the labels, so a labeling boundary is present.

For the purposes of examining differences in behavior within a heuristic pair combination, the meta-controller can also examine the distribution of the labeled training set using `RelPerf` metrics produced by other candidate heuristic pairs not present in the combination. Figure 5.3 contains six such plots. The three plots on the left (Figs. 5.3a, 5.3c,

and 5.3e) show labeled training set problems for the heuristic pair combination E and G plotted against `RelPerf` metrics from three other heuristic pair combinations. The three plots on the right (Figs. 5.3b, 5.3d, and 5.3f) show the labeled training set problems for the heuristic pair combination F and J against the `RelPerf` metrics from the same three other heuristic pair combinations.

In Figs. 5.3a, 5.3c, and 5.3e, the label clusters are generally well formed and separated. The opposite phenomenon is seen in Figs. 5.3b, 5.3d, and 5.3f where the label clusters are mostly overlapping. The fact that the labels for the E and G heuristic pair combination continue to exhibit separation even when plotted by the `RelPerf` of different heuristic pairs shows that the labels are responding to intrinsic properties of the training set problems—this is due to differences in heuristic pair behavior between E and G. The actual architecture search algorithms used in E and G support this notion. The architecture heuristic algorithm used by E is a greedy hill-climber with narrow neighborhood filter, an exploitative search procedure. The architecture search algorithm used by G is opportunistic hill-climber with a wide neighborhood filter, an exploratory search procedure.

By measuring the degree of overlap between data sets, the meta-controller can assess behavior differences between two heuristic pairs. The following procedure was developed in support of the meta-controller to measure the degree of overlap between data sets.

1. Determine the center, C_i, of each data set, $\mathcal{S}_i$.

2. Compute $\vec{C}_{1,2}$, a vector from C_1 to C_2.

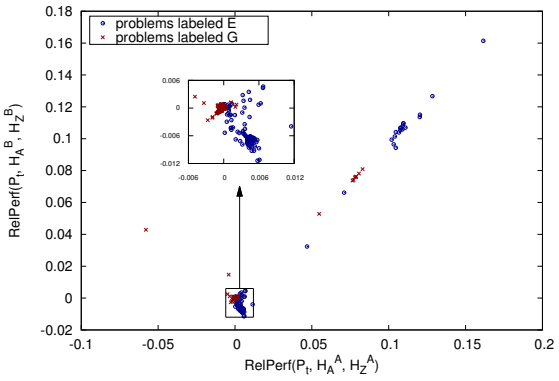
3. For each point, $p_j \in (\mathcal{S}_1 \cup \mathcal{S}_2)$, compute $\vec{p}_j$, a vector from p_j to C_1.

4. Project all p_j onto $\vec{C}_{1,2}$ using $\vec{p}_j$.

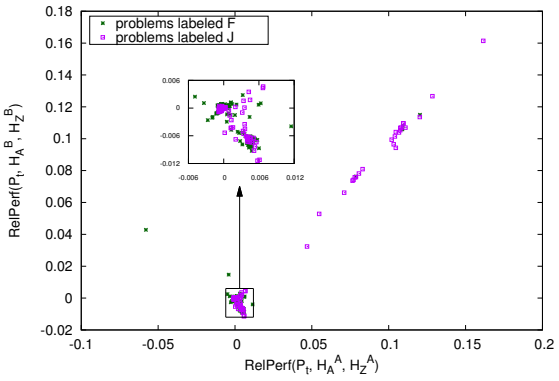
5. For each projected point, $\acute{p}_j \in (\acute{\mathcal{S}}_1)$, compute the fraction of $\acute{p} \in (\acute{\mathcal{S}}_2)$ between $\acute{p}_j$ and $C1$.

6. For each $\acute{p}_k \in (\acute{\mathcal{S}}_2)$, compute the fraction of $\acute{p} \in (\acute{\mathcal{S}}_1)$ between $\acute{p}_k$ and $C2$.

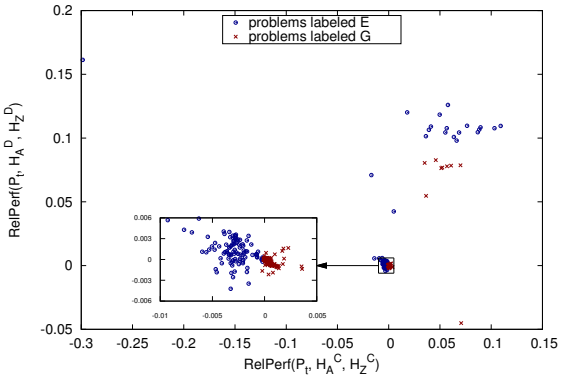
7. Calculate the average fraction for all $\acute{p}$—this is the `Overlap` metric measuring the degree of overlap between data set 1 and data set 2.



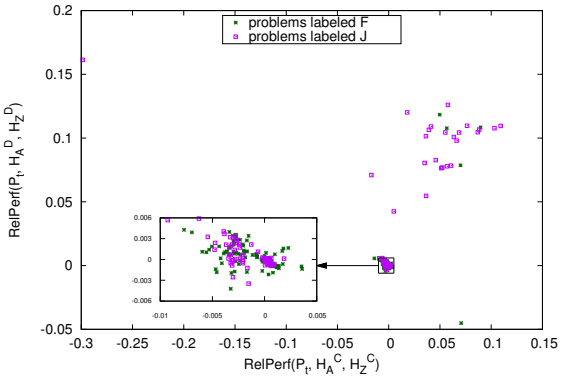
(a) E and G plotted with `RelPerf` of A and B.



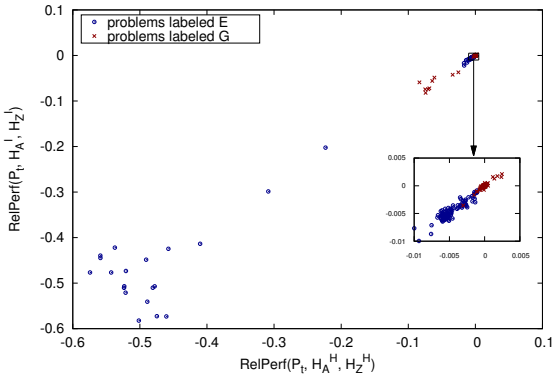
(b) F and J plotted with `RelPerf` of A and B.



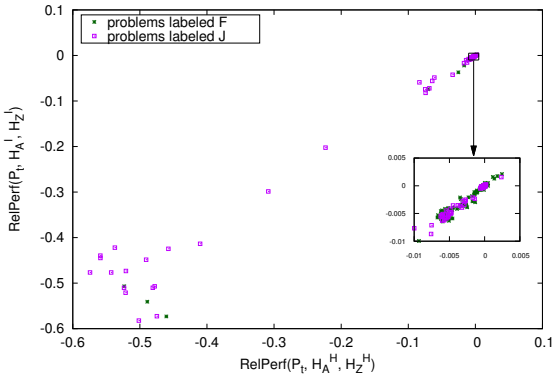
(c) E and G plotted with `RelPerf` of C and D.



(d) F and J plotted with `RelPerf` of C and D.

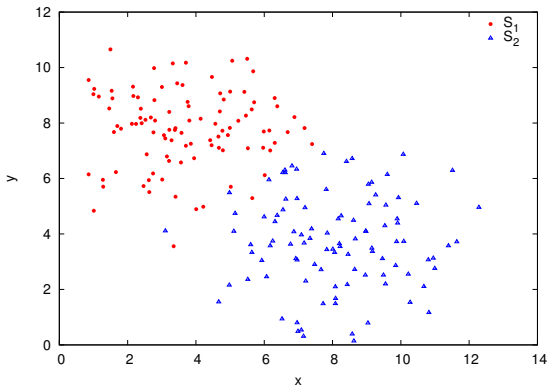


(e) E and G plotted with `RelPerf` of H and I.

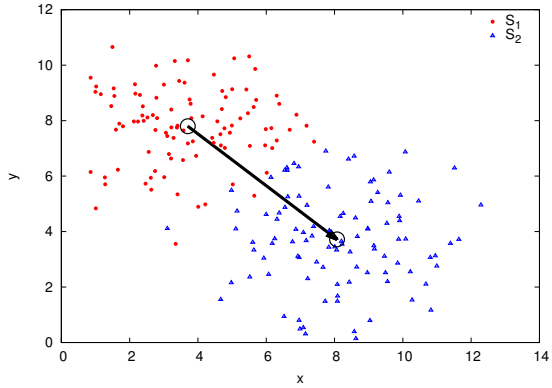


(f) F and J plotted with `RelPerf` of H and I.

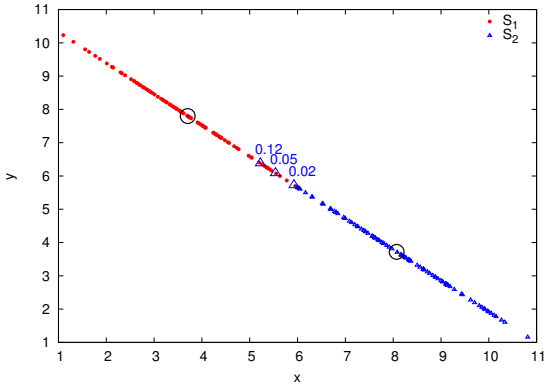
Figure 5.3: In Fig. (a), (c), and (e), the entire training set is labeled with either heuristic pair E or heuristic pair G. In Fig. (b), (d), and (f), the entire training set is labeled with either heuristic pair F or heuristic pair J. Indications of behavior differences in the heuristic pairs can be observed by plotting with the `RelPerf` of other heuristic pairs.



(a) Scatter plot of Gaussian data sets.



(b) Computation of $\vec{C}$.

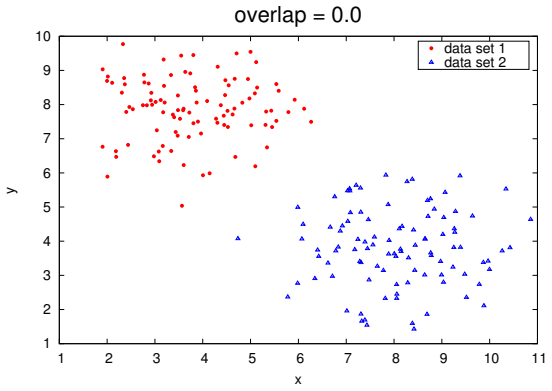


(c) Projection of $\acute{p}$ on $\vec{C}$.

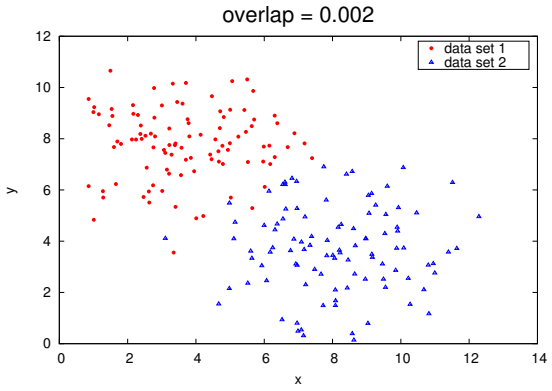
Figure 5.4: Figures (a), (b), (c) demonstrate how the `Overlap` metric is calculated. Figure (a) shows two randomly generated Gaussian data sets, $\mathcal{S}_1$ and $\mathcal{S}_2$. In Fig. (b), $\vec{C}_{1,2}$ is computed. Figure (c) shows the projection of the data sets onto $\vec{C}_{1,2}$. Three of the projected points in $\acute{\mathcal{S}}_2$ are labeled with the fraction of $\acute{\mathcal{S}}_1$ between them and the center of $\mathcal{S}_2$.

A graphical depiction of this process is shown in Fig. 5.4. Figure 5.5 demonstrates how the `Overlap` metric increases as two data sets become less separated. The `Overlap` metric can be computed in an n-dimensional space.

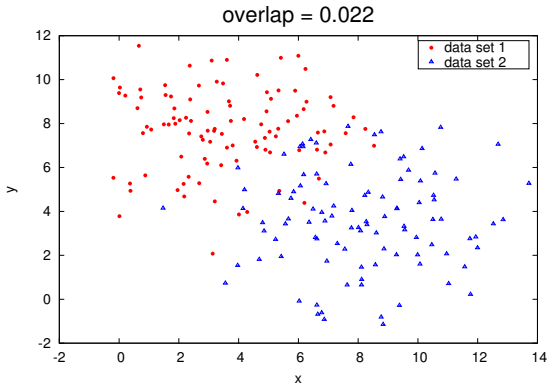
When calculating the `Overlap` metric for a heuristic pair combination, $\mathcal{S}_1$ is the set of training problems labeled with heuristic pair j and $\mathcal{S}_2$ is the set of problems labeled with heuristic pair k. The problems are plotted in an n_c-dimensional space where n_c is the number of candidate heuristic pairs. The value used for i-th dimension is the `RelPerf` metric of heuristic pair i on the given problem.



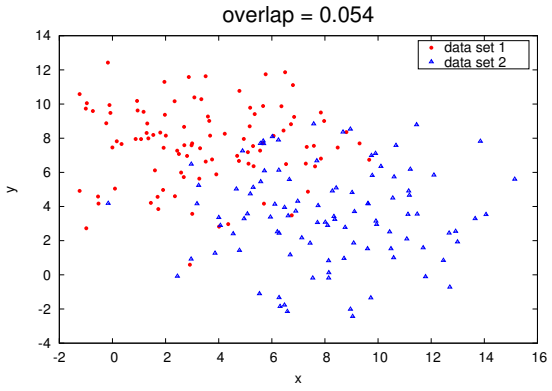
(a) $\sigma = 1.0$



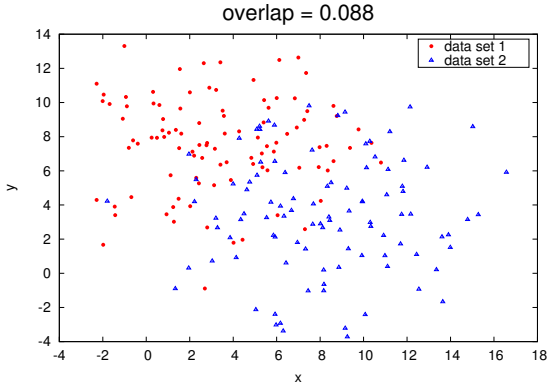
(b) $\sigma = 1.5$



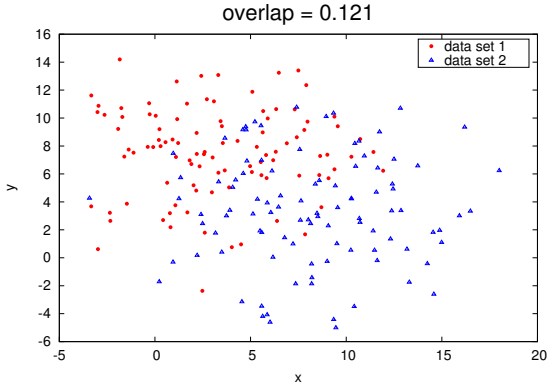
(c) $\sigma = 2.0$.



(d) $\sigma = 2.5$.



(e) $\sigma = 3.0$



(f) $\sigma = 3.5$

Figure 5.5: Figures (a) - (f) show two randomly generated Gaussian data sets. The `Overlap` metric increases as the data sets become less separated.

The `Overlap` metric is converted into a score as follows:

$$\texttt{OverlapScore}((\mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z), (\mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z)) \;=\; \frac{1}{\texttt{Overlap}((\mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z), (\mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z))}. \tag{5.13}$$

The `OverlapScore` is the final component of the `AggregateScore` that will be used to determine the best combination of heuristic pairs.

Find Suitable Pairs: Aggregate Assessment Sub-Step

The meta-controller combines its assessments of the potential performance, the balance of the training set, and the behavioral differences using the `AggregateScore` which is computed as follows:

$$\begin{aligned}\texttt{AggregateScore}((\mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z), (\mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z)) = &\left(\texttt{BalanceScore}((\mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z), (\mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z))\right)^{w_b} \\ &\times \left(\texttt{PotentialGainScore}((\mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z), (\mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z))\right)^{w_p} \\ &\times \left(\texttt{OverlapScore}((\mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z), (\mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z))\right)^{w_o}\end{aligned} \tag{5.14}$$

where w_b, w_p, and w_o are weights for the `BalanceScore`, `PotentialGainScore`, and `OverlapScore` respectively. The heuristic pair combination with the highest `AggregateScore` on the training set is then used in the subsequent steps of training of the SVM.

In practice, the settings $w_p = 0.25$, $w_b = 0.375$, $w_o = 0.375$ were found to be effective by applying this weighting methodology to the selection of heuristic pair combinations. The top aggregate scoring heuristic pair combinations were then used to build label training sets for the SVM. After studying the SVM's performance, the results were used to adjust the weighting methodology. This iterative process resulted in the selection of these weight values.

Remove Noisy Data

For some training problems, the label determined in Equation 5.7 may not have statistical significance—such training problems are considered noisy problems. The SVM meta-controller's performance may be harmed if noisy problems are included in the training set—the SVM may alter the positioning of the hyperplane to accommodate these training problems, which can negatively impact the effectiveness of the decision rule generated by the SVM.

For each problem, $\mathcal{P}_t$, in the training set, the t-test is used to compare `result tuples` that match $(\mathcal{P}_t, \mathcal{H}^j_{\mathcal{A}}, \mathcal{H}^j_Z)$ to result tuples that match $(\mathcal{P}_t, \mathcal{H}^k_{\mathcal{A}}, \mathcal{H}^k_Z)$. If the t-test shows no statistical significance at the $\alpha = 0.01$ level, the problem is not used for either parameter selection or training the SVM.

Determine Penalty Weights

One of the advantages of using SVM over KNN is that it enables the use of penalty weights (see Section 2.4.2). Penalty weights can be used to address two problems faced by the meta-controller:

1. unbalanced training sets and
2. asymmetric cost of misclassification.

Penalty Weights for Unbalanced Data Sets

As discussed previously, unbalanced training sets present challenges for SVM classification that can be partly addressed through the use of penalty weights. Raskutti et al. in [109] employs penalty weights to address unbalanced data sets with some success. However, the authors do not offer a rationale for the simple formulas used to determine the weights.

In developing a penalty weight procedure for the meta-controller, the assumption made was that SVM misclassifies at a high rate on unbalanced training sets due to asymmetries in the uncertainty of the population *edges*. Here *edge* is taken to mean the portion of the population close to the edge of the largest margin computed by the SVM. The SVM

algorithm does not model the underlying populations from the samples and then fit a largest-margin hyperplane between the modeled populations. Rather, it considers the data it is given (i.e., the training set) directly, and then fits the largest-margin hyperplane possible between the sample sets.

On unbalanced training sets, the *edge* of the smaller sample set will usually be less dense than the *edge* of the larger sample set. If the kernel trick (see Section 2.4.2) is employed and the training set is projected into a higher dimensional space to create linear separability, the density of both *edges* are further reduced, and the SVM may effectively try to push a hyperplane into the gaps between the points of the smaller set. This phenomenon results in the SVM overfitting the contours of the smaller sample set, but not the larger set. Overfitting in SVM is controlled by the parameter C. Recall Equation 2.18 from Section 2.4.2, reprinted here for convenience.

$$C \sum_{n=1}^{N} \xi_n + \frac{1}{2} \|w\|^2 . \tag{5.15}$$

The value ξ_n is called a slack variable and measure the degree to which some of the points (the support vectors) have crossed into the margin boundary (for points that are not support vectors ξ is equal to 0) [8]. The unit of measurement for ξ_n is the margin width. The value C is essentially an inverted regularization coefficient and is used to prevent overfitting of the data. If the value of C is too high, it may lead to overfitting the data. In the case of small, unbalanced training sets, the overfitting is one-sided—penalty weights allow essentially two different C parameters:

$$C_i = C \times w_i, \qquad i \in 0, 1 \tag{5.16}$$

where i is the class and w_i is the penalty weight for that class [17]. If properly implemented, lowering C for the larger sample set while raising C for the smaller sample set will mitigate the risk of one-sided overfitting.

In building an effective SVM for the meta-controller, penalty weights were used to inform the SVM of the greater uncertainty in the smaller data set. Note that in the SVM's final convergence on a hyperplane, the SVM is considering only the subset of points that have a

reasonable chance of being support vectors. To express the uncertainty of the position of this subset, a confidence interval measured in units of the margin width is used.

$$\overline{x}_i \ \pm \ t_{[1-\alpha/2,((f_i n_i)-1]} \frac{s_i}{\sqrt{f_i n_i}} \tag{5.17}$$

where $t_{[1-\alpha/2,((f_i n_i)-1]}$ is the critical value of the t distribution, n_i is the number of samples labeled with i, f_i is the fraction of samples residing close enough to the margin to affect its width and/or alignment, and s_i is the standard deviation of n_i measured in units of the margin width.

The following expression shows class i's preferred margin width:

$$\acute{m}_i = \max\left(\left(\ \frac{m}{100} \ \right) \ , \ \left(m \ - \ t_{[1-\alpha/2,((f_i n_i)-1]} \frac{s_i}{\sqrt{f_i n_i}} \right) \ \right) \tag{5.18}$$

where m is the original margin width. From Equation 5.15, it is known that there is an inverse relationship between the margin width and the penalty weights. Class i's preferred penalty weight for uncertainty can be expressed as:

$$\acute{w}_{u,i} = \frac{1}{\acute{m}_i}. \tag{5.19}$$

The weights are then normalized (this allows the bounds on C to remain in the search for optimal kernel parameters later on) which provides the penalty weight contribution for uncertainty.

$$w_{u,i} = \frac{2\acute{w}_{u,i}}{\acute{w}_{u,0} + \acute{w}_{u,1}} \tag{5.20}$$

This method does have the drawback that SVM practitioners may have trouble specifying some of the key inputs. The margin width is not generally known a priori, so describing the standard deviation in terms of the margin requires a best guess. If overfitting of the smaller data set is occurring without penalty weights then a standard deviation of 0.5 margin widths is a reasonable initial estimate; this value was used in the SASSY SVM

meta-controllers.

SVM practitioners may also have difficulty specifying the fraction of the population residing in the *edge*. Estimating this value will require considering:

- the expected number of support vectors in the produced SVM model,
- the dimensionality of the sample problems, and
- the level of model complexity providing the highest accuracy.

Complex models on multi-dimensional data may have a significant fraction of the sample set in the *edge*. For the SASSY SVM meta-controller, the fraction of the sample set in the *edge* was estimated to be 0.8 because of the high dimensionality of $\mathcal{F}(\mathcal{P})$ and because complex models tend to provide the best performance on SASSY meta-controller training sets.

Finally, the practicioner must specify the α value. The α parameter is conducive to tuning the procedure and generally can range from 0.01 to 0.1. Smaller values of α further favor the smaller sample sets. The SASSY SVM meta-controller was found to be effective with $\alpha = 0.05$, which was sufficiently low to limit the likelihood of overfitting the smaller data set.

Penalty Weights for Asymmetric Cost

On many pattern recognition problems, the costs of misclassification are asymmetric—that is the cost of incorrectly classifying class 0 as class 1 is different from the cost of incorrectly classifying class 1 as class 0. An examination of a `RelPerf` plot can help determine if the costs of misclassification are symmetric or asymmetric in the pattern recognition problem faced by the SASSY meta-controller. The `RelPerf` plot in Fig. 5.6 shows the computed `RelPerf` values for heuristic pairs E and H (see Table 4.4 in Section 4.7.2) on sample SASSY optimization problems in a training set. The points labeled with heuristic pair H are quite close to the label boundary where the `RelPerf` values are equal. While the majority of points labeled heuristic pair E are also close to the label boundary, a significant fraction form a cluster a large distance away from the label boundary. This plot clearly shows an asymmetric cost in the case of heuristic pair combination E and H.

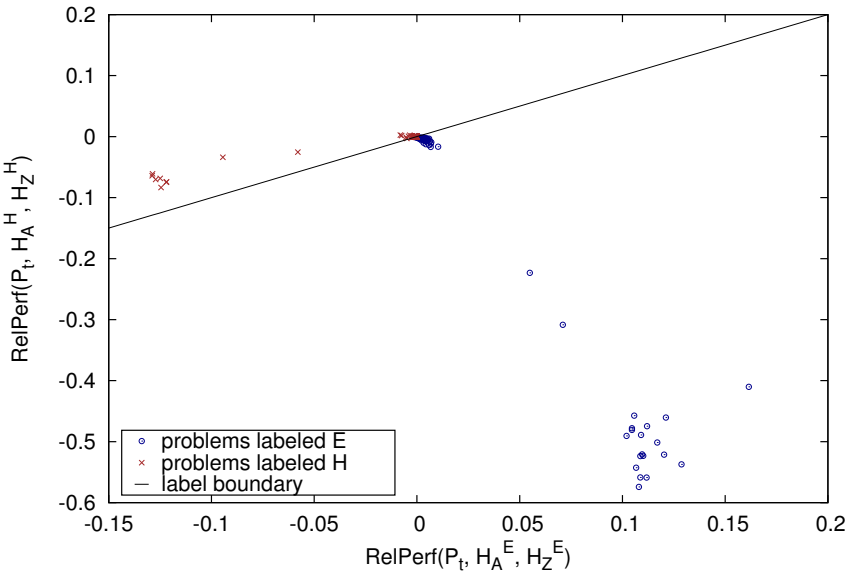


Figure 5.6: E and H plotted with `RelPerf` of E and H.

The cost associated with misclassifying a problem as k when the true label is j can be determined as follows:

$$c_j = \frac{1}{n_j} \sum_{i=1}^{n_j} \left(\texttt{RelPerf}(\mathcal{P}_i^j, \mathcal{H}_\mathcal{A}^j, \mathcal{H}_Z^j) - \texttt{RelPerf}(\mathcal{P}_i^j, \mathcal{H}_\mathcal{A}^k, \mathcal{H}_Z^k) \right). \tag{5.21}$$

The cost associated with misclassifying a problem as j when the true label is k is similar:

$$c_k = \frac{1}{n_k} \sum_{i=1}^{n_k} \left(\texttt{RelPerf}(\mathcal{P}_i^k, \mathcal{H}_\mathcal{A}^k, \mathcal{H}_Z^k) - \texttt{RelPerf}(\mathcal{P}_i^k, \mathcal{H}_\mathcal{A}^j, \mathcal{H}_Z^j) \right). \tag{5.22}$$

Now the costs must be converted into penalty weights for cost, w_c. To avoid changing the bounds of C in the search for optimal kernel parameters requires that $0 < w_c < 2$.

Equations 5.23 and 5.24 accomplish this.

$$w_{c,j} \;=\; 2\left(\frac{\frac{c_j}{c_k}}{1+\frac{c_j}{c_k}}\right) \tag{5.23}$$

$$w_{c,k} \;=\; 2 - w_{c,j} \tag{5.24}$$

Finally, the penalty weights for uncertainty and cost are combined.

$$w_j \;=\; w_{u,j} \;\times\; w_{c,j} \tag{5.25}$$

$$w_k \;=\; w_{u,k} \;\times\; w_{c,k} \tag{5.26}$$

These final weight values are provided to the `libsvm` SVM library [17] when the meta-controller calls procedures for both cross-validation and training.

Optimize SVM Parameters

As discussed in Section 2.4.2, SVM performance can vary dramatically depending on the values selected for kernel parameters and C. The kernel parameters control the possible levels of model complexity, while C is used to prevent overfitting the data. The SASSY SVM meta-controller uses the *radial basis function* kernel (see Section 2.4.2 for more information). The radial basis function kernel has one parameter, γ.

When using the radial basis function kernel, the authors of `libsvm` recommend using their grid search tool to determine the optimal values of γ and C [54]. This tool (more fully described in Section 2.4.2) plots accuracy bands around different ranges of γ and C values. The accuracy value for each point (γ, C) is produced from an n-fold cross-validation process (the user specifies the value of n). The user can then choose to repeat the process focusing with higher resolution on the most promising regions in the parameter space. This process can be time-consuming.

During the initial development of the offline SVM meta-controller, the `libsvm` grid search tool was used extensively to find the optimal values for γ and C. The accuracy landscapes generated by the cross-validation procedure were highly unusual. There was a

degree of self-similarity in the structure of the many optima. At each level of zoom, multiple optima appeared. At this stage in the meta-controller's development, the need for penalty weights to compensate for unbalanced training sets was not yet known, which may have contributed to this strange accuracy landscape. The absence of multiple optima can be noted in Fig. 5.7, which shows grid search results when the penalty weight procedure for unbalanced data sets has been applied.

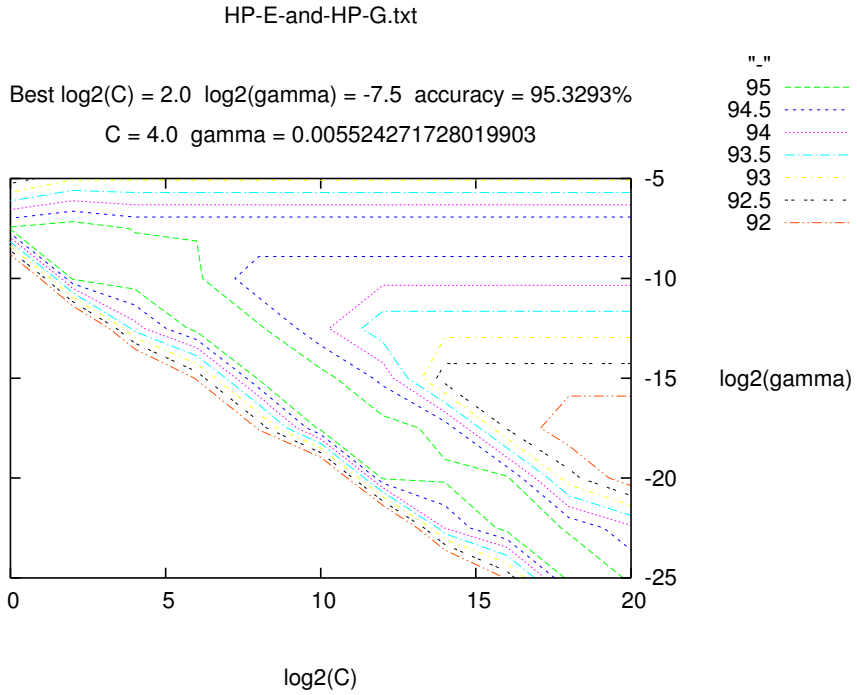


Figure 5.7: Refining search with grid search tool from LibSVM.

Use of the `libsvm` grid search tool requires substantial human interaction to find optimal values of γ and C. The strange landscape with multiple optima multiplied the effort required. To introduce automation into the process, the grid search tool was replaced with a genetic algorithm (GA). When properly employed, GAs have proven effective in similar difficult landscapes [24]. Additionally, the development of an automated procedure for tuning these parameters enabled the creation of an online SVM meta-controller.

A common characteristic of all SASSY meta-controller accuracy landscapes is that at

Parameter	Minimum	Maximum
$log(C)$	-10.0	40.0
m	-3.0	0.5
b	-60.0	20.0
$log(\gamma)$	-40.0	10.0

Table 5.2: Minimum and maximum values in the GA search for SVM parameters.

the highest levels of the search there is a linear negative correlation between $log(C)$ and $log(\gamma)$ (see Fig. 5.7). This relationship is common to many classification problems using the radial basis function kernel and can be seen in the grid searches performed in [54]. The presence of this relationship was used to speed early convergence of the GA by encoding $log(\gamma)$ as a linear function of $log(C)$:

$$log(\gamma) \; = \; m \; log(C) \; + \; b \tag{5.27}$$

where m is the slope and b is the $log(C) = 0$ intercept. Thus, the GA encoded three parameters, $log(C)$, m, and b. The number of bits for each of these three parameters was set to 29. The maximum and minimum values for each parameter can be found in Table 5.2; the values for $log(C)$ and $log(\gamma)$ are those suggested in [54]. The values for m and b are drawn from analysis of parameter optimization plots for meta-controller SVMs.

The parent and offspring size were each set to 20. Parent selection was conducted with linear rank selection described in Equation 2.15 with the selection pressure set to 1.0. The objective function used to rank the parents was the observed weighted accuracy from the cross-validation. The bit-flip probability was set to $\frac{1}{L}$ where L is the length of the binary string (in this case $\frac{1}{87}$). The crossover probability was set to 0.15. The value for n in the n-fold cross-validation procedure was set to 50.

For the offline SVM meta-controller, the GA search for C and γ was run 20 times for 12 hours each time. The number of repetitions turned out to be unnecessary as each search converged to similarly peforming optima each time. The `libsvm` library is safe for multi-threading, so the GA search ran with 8 threads to maximize resource usage on systems with

dual quad-core CPUs.

Train SVM Model

Once the previous procedures have been completed, training the SVM model is trivial. The SVM algorithm is provided with the SVM search parameters, C and γ. The penalty weights for each class are specified at the same time, and the scaled training set is loaded into a specific C programming language struct specified in the `libsvm` documentation. Training the model on sets with a thousand SASSY sample training sets takes about a second on an ordinary modern personal computer. The output of the `libsvm` training process is a decision boundary that can be used to classify the problems encounted by the meta-controller.

Applying the SVM Model

Applying the decision rule generated by the SVM training process to an encountered optimization problem, $\mathcal{P}_e$ is relatively simple:

1. calculate the features $\mathcal{F}(\mathcal{P}_e)$,
2. scale the features $\mathcal{F}(\mathcal{P}_e)$ by the same factors as the training set,
3. apply the kernel function to transform the scaled $\mathcal{F}(\mathcal{P}_e)$ (performed by `libsvm`), and
4. apply the decision rule to the scaled, transformed $\mathcal{F}(\mathcal{P}_e)$.

The output of the decision rule is the label of the heuristic pair, $(\mathcal{H}_\mathcal{A}, \mathcal{H}_Z)$, to use on the given problem. The meta-controller uses the label to look up the appropriate pair, and informs the autonomic controller to use the heuristic algorithms in the pair.

5.3.6 Online Training SVM Meta-Controller

The development of the online training SVM meta-controller was motivated by the considerable overhead involved in developing an offline training set, finding an appropriate heuristic pair combination, and optimizing the SVM parameters needed to train the SVM. This can be time consuming and requires significant computing resources. During the time that the offline training SVM meta-controller is under development, the administrator must decide

whether to wait for the meta-controller before deploying the application or whether to run the application with a simple autonomic controller. The former choice involves substantial delay and the latter introduces additional complexity and burden on the administrator.

Another approach is to implement the SVM meta-controller in a fashion similar to the `Overall Best` and `Context Best` KNN meta-controllers. One of the differences between the KNN meta-controller and an online SVM meta-controller is that the KNN meta-controller has an implicit model after encountering, collecting, and studying its first optimization problem. The online SVM meta-controller will need to have collected a significant number of training problems to create a reasonable model.

Until the training set has reached a minimum threshold size, the online SVM meta-controller mimics the behavior of the `Overall Best` meta-controller. In the work presented here, the minimum number of training problems to train an SVM model was set to 50 (this is before filtering out noisy problems, so the actual number of problems presented to the SVM is likely to be smaller).

After studying the 50th collected problem, the online SVM meta-controller uses the same `AggregateScore` assessment procedure as the offline SVM meta-controller for selecting a heuristic pair combination. During testing, the online SVM meta-controller working on the SAS-65 application (described in Section 6.3.1) typically took less than 3.2 seconds (and usually less than 1 second) to consider 28 different heuristic pair combinations on a 2.4 GHz CPU.

Once the heuristic pair combination is selected, the online SVM meta-controller uses a slightly different GA for selecting the SVM parameters. This search is run just one time for each SVM training. The bit-flip and crossover probabilities are unchanged from the offline SVM meta-controller. Cross-validation is the same as the offline SVM meta-controller (50 folds). For the online SVM meta-controller, the GA's population size has been reduced from 20 to 10, and the 12-hour search limit is replaced by a search limit of 1,000 $(log(C), log(\gamma))$ evaluated points. Most of these searches took under an hour to complete with dual quad-core 2.4 GHz CPUs.

After successfully training the SVM model, the online SVM meta-controller uses the newly developed SVM decision rule to classify optimization problems. Whenever five new

problems have been collected, the online SVM meta-controller uses the new data to revisit its selection of heuristic pair combination and SVM parameter selection. The online SVM meta-controller reverts to `Overall Best` if no training problems remain for one of the labels after noisy training problems have been removed.

To help reduce some of the training burden, the online SVM generates result tuples for candidate heuristic pairs on each collected problem, $\mathcal{P}_t$, in a round-robin fashion. Starting with the third round, following the completion of a round, the online SVM meta-controller follows a three-step process to winnow down the number of heuristic pairs tested against $\mathcal{P}_t$.

1. Determine the current best heuristic pair, $(\mathcal{H}^c_{\mathcal{A}}, \mathcal{H}^c_Z)$ on $\mathcal{P}_t$, from the candidate heuristic pairs.

2. Compare each of the other i candidate heuristics to $(\mathcal{H}^c_{\mathcal{A}}, \mathcal{H}^c_Z)$ using the t-test with $\alpha = 0.01$.

3. If a statistically significant difference is found between $(\mathcal{H}^i_{\mathcal{A}}, \mathcal{H}^i_Z)$ and $(\mathcal{H}^c_{\mathcal{A}}, \mathcal{H}^c_Z)$, then cease calculating result tuples for $(\mathcal{H}^i_{\mathcal{A}}, \mathcal{H}^i_Z)$ on problem $\mathcal{P}_t$.

This modification saves the meta-controller from wasting resources analyzing clearly inferior heuristic pairs on $\mathcal{P}_t$.

5.4 Concluding Remarks Regarding Meta-Controller Approaches

This chapter has described in detail four different approaches for meta-controllers in SASSY. The `Overall Best` meta-controller simply monitors which heuristic algorithm pair provides the best expected performance across all encountered optimization problems.

The three `Context Best` meta-controller approaches analyze the encountered optimization problem and then attempt to predict the best heuristic algorithm pair for solving the encountered problem. All three `Context Best` meta-controllers conduct the same feature analysis of the encountered optimization problems. The first `Context Best` meta-controller uses a vanilla KNN machine learning algorithm. The second and third `Context Best` meta-controller methods use SVM for the machine learning algorithm. The difference between

the SVM `Context Best` meta-controllers is that one trains offline while the other trains online.

Some interesting techniques were developed in support of the SVM meta-controllers. A technique for measuring the degree of overlap between two populations of data points was presented. This technique for measuring overlapping populations could be extended by incorporating the k-means clustering algorithm [58], which could lead to very precise measurements of population overlap. To address problems in working with unbalanced training sets, a rationale and a technique were developed for setting penalty weights in SVM. The next chapter will provide experimental evaluation of the autonomic controllers presented in Chapter 4 and the meta-controllers presented in this chapter.

Chapter 6: Experimental Evaluation

This chapter presents the experimental evaluation of the SASSY autonomic controller and meta-controllers described in Chapters 4 and 5. The first section describes the implementation of the SASSY autonomic controller and meta-controllers. The second section provides the simulation design used by the experiments. The third section examines the development of the SASSY test applications. The scalability experiments and their results are covered in the fourth section, while the fifth section details the meta-optimization experiments. The sixth section contains the experimental results for three different `Context-Best` meta-controllers.

6.1 Autonomic Controller Implementation

The performance monitor, the autonomic controller, the meta-controllers, and the heuristic search algorithms were written in C++ using the GNU C++ compiler. The source code has been compiled and tested on both Linux (CentOS version 5.11) and Windows Vista using Intel compatible processors. The *libsvm* library [17] was used for the SVM meta-controller.

The heuristic search algorithms were written as C++ templates—this allows the same search logic to be applied to both architecture and service selection. The genetic algorithm (GA) code used for both the meta-optimization search described in Section 4.7 and the SVM parameter search described in Section 5.3.5 for the SVM parameter search also utilized these heuristic search code templates. The heuristic search algorithms employed multi-threading by using the BOOST thread library [128]. Random number streams used to produce stochastic behavior in the heuristic search algorithms were provided using the Mersenne Twister [70, 84].

6.2 Simulation Design

The experiments described in this section use a simulated service environment. This simulation generates changes to the service environment that will force the autonomic controller to re-architect multiple times over the course of an experiment. The response variable in these experiments is the mean U_g collected by the SASSY performance monitor.

6.2.1 Simulated Services

Simulated SPs were developed for evaluating the autonomic controller and meta-controllers. These simulated SPs advertise their service time, capacity in transactions per second, and availability. To test the ability of the autonomic controller to react and adapt to service failures, the service instances must demonstrate failure behavior in simulation. Four performance variables govern the behavior of an SP:

- s (mean service time),
- c (capacity),
- t_f (mean time to failure), and
- t_r (mean time to repair).

The availability of the SP is derived from t_f and t_r.

Each simulated SP has two state variables, $CONTRACT$ and $STATUS$, that together define six distinct states. The variable $CONTRACT$ is a boolean that designates whether the SP is included in the current SASSY service selection. The $STATUS$ variable has three possible values: *Normal*, *Soft Fail*, and *Hard Fail*. The states and their corresponding variable settings are shown in Table 6.1. A state transition diagram for an SP is shown in Fig. 6.1. All SPs initially start in the *Idle, Normal* state. State transitions due to the autonomic controller changing the service selection are denoted by a thick line in Fig. 6.1. The dashed lines show state transitions resulting from failure events.

Whenever a simulated SP arrives in a normal state (including the initial start), the SP schedules a random failure event using an exponential distribution with a mean equal to t_f. A coin flip is used to determine if the failure is a hard failure or a soft failure. Whenever an

State Name	*CONTRACT*	*STATUS*	Description
Idle, Normal	FALSE	NORMAL	unused, operating normally
Contract, Normal	TRUE	NORMAL	in service selection, operating normally
Idle, Soft Failure	FALSE	SOFT FAIL	unused, operating poorly
Contract, Soft Failure	TRUE	SOFT FAIL	in service selection, operating poorly
Idle, Hard Failure	FALSE	HARD FAIL	unused, unresponsive
Contract, Hard Failure	TRUE	HARD FAIL	in service selection, unresponsive

Table 6.1: State descriptions for simulated SPs.

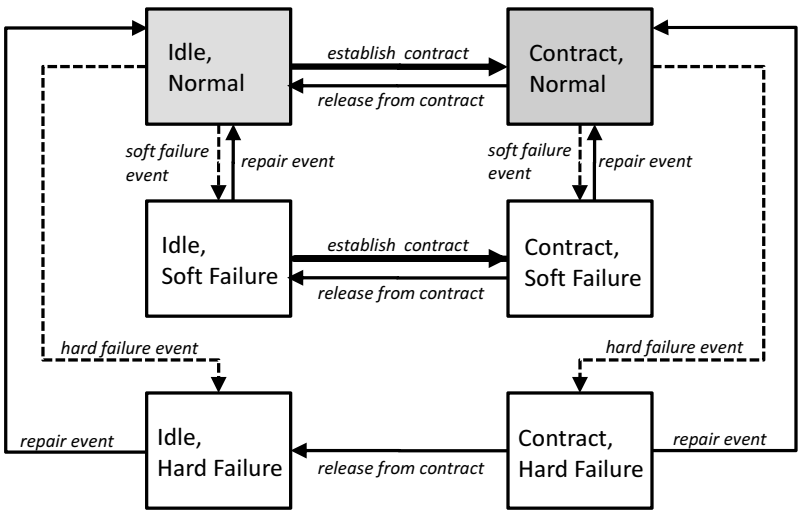


Figure 6.1: State transition diagram describing the behavior of an SP.

SP arrives in a failure state, the SP schedules a random repair event using an exponential distribution with a mean equal to t_r.

The SPs can simulate their function at various security levels with specified performance effects. When a security feature is selected in a contract, the appropriate multipliers corresponding to the selected security option and level are applied to s and c.

6.2.2 Simulation Detail

Each simulation experiment is divided into 500 discrete intervals called *controller intervals* of duration ϵ time units. The value t_r was set to 1 time step for each SP in the simulation. The value t_f was derived from the SP's advertised availability.

For the purpose of the simulation, the re-architecting search and the implementation of a new architecture ($\mathcal{A}$) and service selection (Z) are assumed to be instantaneous. In reality, the re-architecting searches take approximately 5 to 20 seconds and implementing the new architecture and service selection might take up to a minute.

At each time step, the simulation uses analytic performance models similar to those described in Section 4.3 to assess the current performance of the system. The performance monitor collects these values and evaluates the U_g of the SASSY application according to the provided global utility function. If the global utility falls below the autonomic controller's re-architecting threshold (set to 80% of the predicted U_g of the current $\mathcal{A}$ and Z), a re-architecting search is initiated. If a previously failed service is repaired, this also triggers a re-architecting search. If no re-architecting search occurs, the performance monitor records the value of U_g for the current time step. If the re-architecting search finds a superior $\mathcal{A}$ and Z, the new $\mathcal{A}$ and Z are implemented. The performance monitor collects updated performance measurements and recomputes and records U_g for the current time step.

6.3 Development of SASSY Test Applications

Test SOA applications were generated in a 6-step process:

1. Create an SAS.

2. Determine available security options.

3. Generate available SPs.

4. Generate SSSes.

5. Generate global utility function.

6. Find a starting architecture and service selection.

6.3.1 SASes

SASSY applications were developed at four different levels of scale. An SAS was manually developed with 15 activities (see Fig. 6.2), 25 activities (see Fig. 6.3), 40 activities (see Fig. 6.4), and 65 activities (see Fig. 6.5).

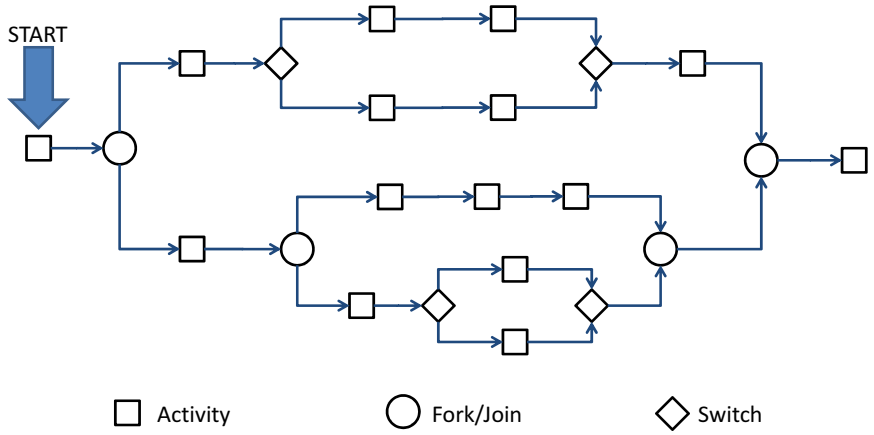


Figure 6.2: SOA application with 15 activities.

6.3.2 Generation of Service Providers

A group of between three to ten SPs are produced for each service activity shown in the SAS. The first step in generating the SPs is to generate a baseline profile for each service

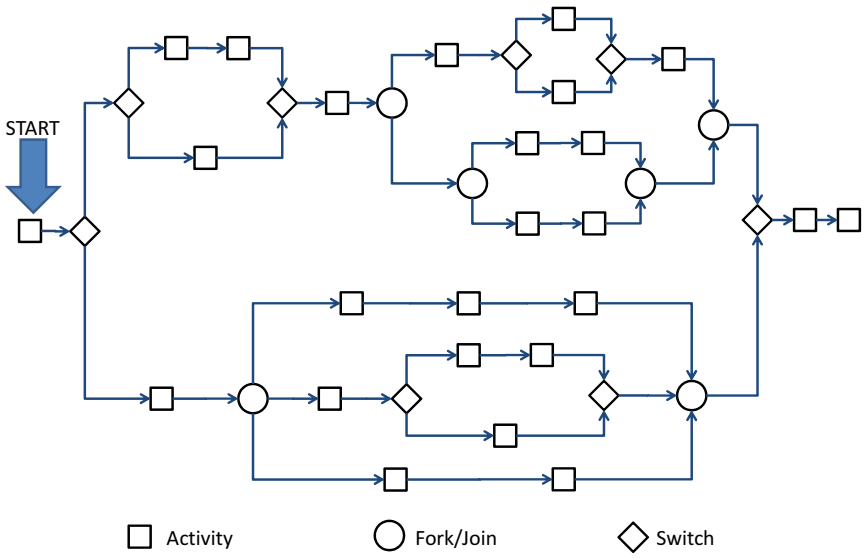


Figure 6.3: SOA application with 25 activities.

activity. The baseline profile, B_i, for a service activity i is randomly determined in the following manner.

- The baseline profile execution time, B_i^E, is set to a random variable taken from a uniform distribution ranging from 1.0 to 50.0 ms.
- The baseline profile capacity, B_i^C, is set to a random variable taken from a uniform distribution ranging from 10.0 to 30.0 requests per second.
- The baseline profile availability, B_i^A, is set to 0.995^x where x is a random variable taken from an exponential distribution with a mean of 1.0.
- The baseline profile cost, B_i^c, is set to a random variable taken from a uniform distribution ranging from $10B_i^E$ to $20B_i^E$ in dollars.

Once B_i has been established, an individual SP j for service activity i is generated from B_i using a random variable, x, taken from a uniform distribution ranging from 0.25 to 1.75. The range for x was selected to create a heterogeneous mixture of SPs from B_i. The metrics for SP j are computed as follows (the value for x is computed separately for each metric):

- The execution time, $E_{i,j}$, is set to xB_i^E.

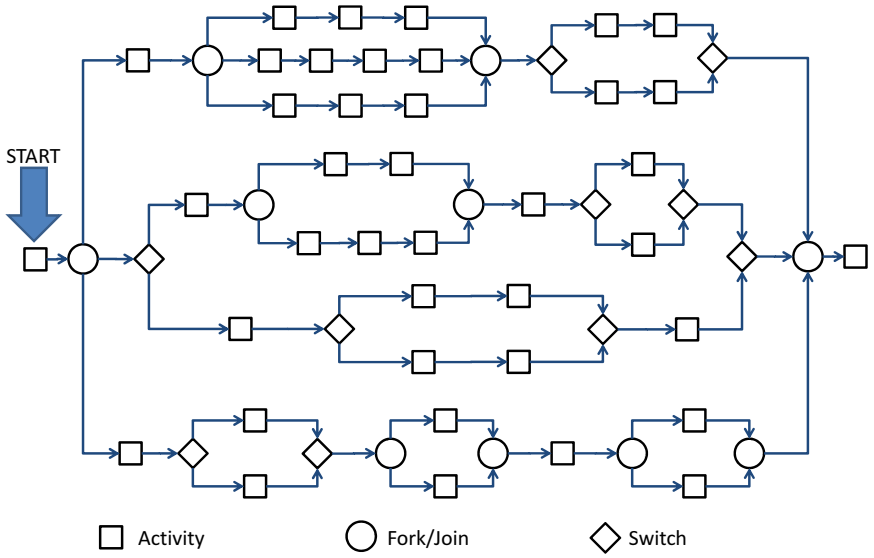


Figure 6.4: SOA application with 40 activities.

- The capacity, $C_{i,j}$, is set to xB_i^C.
- The availability, $A_{i,j}$, is set to $(B_i^A)^x$.
- The cost, $c_{i,j}$, is set to xB_i^c.

Once all of the SPs for a given service activity have been generated, the SP generation algorithm checks to see if any SP is strictly inferior to another SP for the same service activity. SP j is considered strictly inferior to SP k if the following four conditions are met:

$$E_{i,j} > E_{i,k} \tag{6.1}$$

$$C_{i,j} < C_{i,k} \tag{6.2}$$

$$A_{i,j} < A_{i,k} \tag{6.3}$$

$$c_{i,j} > c_{i,k}. \tag{6.4}$$

An SP that is found to be strictly inferior is regenerated. This process repeats until no SP is strictly inferior to another SP with the same service activity.

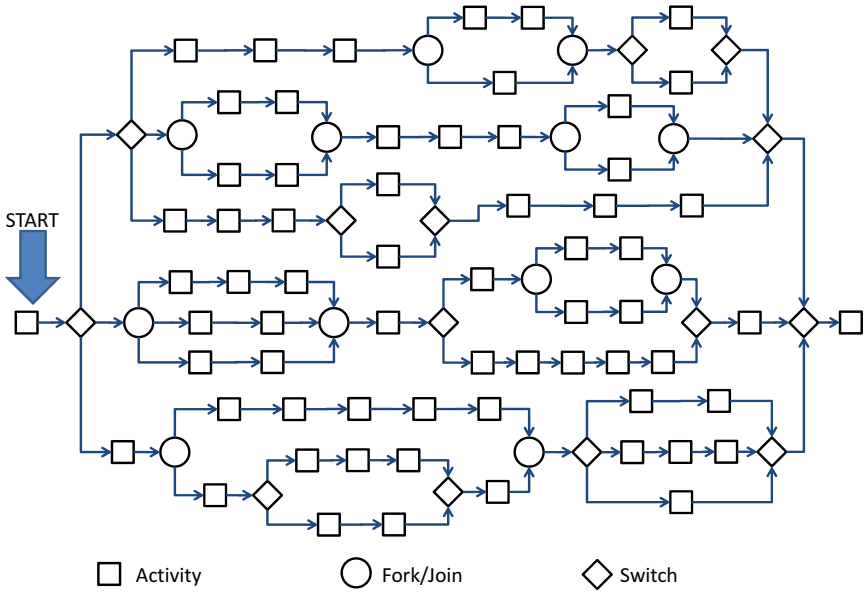


Figure 6.5: SOA application with 65 activities.

6.3.3 Generation of SSSes

As discussed in Section 2.5.3, an SSS is a subset of activities in the SAS that follows a path through the SAS. Each SSS is associated with a metric (e.g., execution time, throughput, availability, or security) and an attribute utility function.

The first two steps in generating an SSS are 1) find the path the SSS takes through the SAS and 2) select the metric for the SSS. The path is determined through a random walk starting at the begining of the SAS graph. When the random walk encounters a switch, it randomly picks one of the edges emanating from the switch. When a fork is encountered, the random walker will proceed down each of the edges emanating from the fork. The random walk concludes when it reaches the final activity on the SAS. No two SSSes are allowed to share both the same metric and the same path.

The metric is randomly determined (each metric has an equal probability of being selected). If the selected metric for the SAS is either execution time, throughput, or availability, a sigmoid function is used for the SSS utility function (see Section 2.2.3). The goal value for the metric in the sigmoid function is denoted by β while the shape of the curve is controlled by α. The value for α (sensitivity parameter) is randomly selected from values

listed in Table 6.2.

Metric	Value 1	Value 2	Value 3	Value 4	Value 5	Value 6
Exe. Time	0.05	0.10	0.15	0.25	0.40	0.65
Throughput	-0.40	-0.25	-0.16	-0.10	-0.06	-0.04
Availability	-2,300	-1,500	-800	-500	-300	-200

Table 6.2: Possible α (sensitivity) values for randomly generated SSSes.

If SSS i uses the execution time metric, the random walk computes the shortest possible expected execution time—this value is stored as $\beta^*_{E,i}$. The value $\beta_{E,i}$ is set to $x\beta^*_{E,i}$ where x is a random variable taken from a uniform distribution ranging from 1.02 to 1.05.

If SSS i uses the throughput metric, the random walk computes the largest possible throughput using the base architecture—this value is stored as $\beta^*_{C,i}$. The value $\beta_{C,i}$ is set to $x\beta^*_{C,i}$ where x is a random variable taken from a uniform distribution ranging from 1.0 to 1.2.

If SSS i uses the availability metric, the random walk computes the highest possible availability using the base architecture—this value is stored as $\beta^*_{A,i}$. The value $\beta_{A,i}$ is set to $\beta^x_{A,i}$ where x is a random variable taken from a uniform distributing ranging from 0.925 to 0.975.

The variance introduced by x allows for the real-world trade-offs that the autonomic controller must make in order to satisfy competing goals (e.g., minimizing execution time while maximizing throughput).

Two security options, each with three levels of information assurance, were made available. The performance impact of the options can be found in Tables 6.3 and 6.4. Increasing the security level of an SP increases the SP's execution time and decreases its capacity. If SSS i uses a security option metric, the utility payout for the highest level of security, U^{high}, is always set to 1.0. The utility payout for the middle level of security, U^{medium}, is randomly determined from a uniform distribution ranging from 0 to 1.0. The utility payout for the

lowest level of security, U^{low}, is randomly determined from a uniform distribution ranging from 0 to U^{medium}. Note that while different SSSes may use the same security option for a metric, each of these SSSes will have its own payout structure.

Metric	Level 0	Level 1	Level 2
Δ Exe. Time	0.00	0.04	0.09
Δ Capacity	0.00	-0.15	-0.30

Table 6.3: Performance impact of security option 1.

Metric	Level 0	Level 1	Level 2
Δ Exe. Time	0.000	0.035	0.080
Δ Capacity	0.00	-0.14	-0.28

Table 6.4: Performance impact of security option 2.

A weighted-geometric utility function is used for the global utility function, $U_g()$, (see Section 2.2.3). The SSS utility functions supply the inputs to $U_g()$. The weights are randomly determined from a uniform distribution ranging from 1.0 to 5.0 to ensure sufficient variance to mimic real-world conditions. For ease of comprehension, the weights are then normalized so that the total weight is 1.0.

A summary of the SSSes produced by this procedure for each of the SASes can be found in Tables 6.5, 6.6, 6.7, and 6.8.

The budget (or cost constraint) for the application is computed as follows:

$$b = 2.25 \sum_{i=1}^{N} \frac{1}{n_i} \sum_{j=1}^{n_i} c_{i,j} \tag{6.5}$$

QoS Metric	Weight	Number of Activities
execution time	0.12	12
throughput	0.04	12
availability	0.15	12
availability	0.12	12
availability	0.12	12
security option 1	0.14	12
security option 1	0.12	12
security option 2	0.19	12

Table 6.5: Summary of SSSes in SAS-15.

QoS Metric	Weight	Number of Activities
execution time	0.08	12
throughput	0.17	11
throughput	0.11	12
throughput	0.05	11
availability	0.17	12
security option 1	0.13	12
security option 2	0.10	13
security option 2	0.07	12
security option 2	0.06	11
security option 2	0.06	11

Table 6.6: Summary of SSSes in SAS-25.

QoS Metric	Weight	Number of Activities
execution time	0.11	26
execution time	0.07	26
execution time	0.07	30
execution time	0.04	26
throughput	0.07	30
throughput	0.03	26
availability	0.16	30
availability	0.06	26
security option 1	0.14	30
security option 1	0.14	30
security option 1	0.07	26
security option 1	0.04	26

Table 6.7: Summary of SSSes in SAS-40.

QoS Metric	Weight	Number of Activities
execution time	0.18	11
execution time	0.03	16
execution time	0.03	11
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
security option 1	0.08	16
security option 1	0.03	9
security option 2	0.11	11
security option 2	0.07	9

Table 6.8: Summary of SSSes in SAS-65.

where N is the number of service activities, n_i is the number of SPs for service activity i, and $c_{i,j}$ is the cost of SP j in service activity i. The factor of 2.25 provides a budget large enough to employ architectural patterns requiring multiple SPs.

6.3.4 Finding Initial Architecture and Service Selection

An initial optimized architecture was found for each SAS; the initial search was conducted with the two different methods shown in Table 6.9.

The search budgets for the initial architecture, $\mathcal{A}_0$, and service selection, Z_0, varied with the size of the SAS (see Table 6.10). Initially, smaller $\mathcal{A}$ search budgets were chosen for the SAS-40 and the SAS-65, but analysis of the search trajectory revealed that the searches had not yet converged. Thus, the searches were rerun with a larger $\mathcal{A}$ search budget to ensure convergence.

6.4 Scalability Experiments

This set of experiments examines the performance of the autonomic controllers and the `Overall Best` meta-controller on all four of the SASSY SASes.

Parameter	Method 1	Method 2
$\mathcal{A}$ search alg.	hill-climbing	evolutionary prog.
$\mathcal{A}$ search mode	opportunistic	overlapping
$\mathcal{A}$ filter j	5	N/A
$\mathcal{A}$ filter k	2	N/A
$\mathcal{A}$ M	N/A	50
$\mathcal{A}$ K	N/A	250
$\mathcal{A}$ step size	N/A	2.0
$\mathcal{A}$ adaptive step factor	N/A	2.0
Z search alg.	hill-climbing	evolutionary prog.
Z search mode	opportunistic	overlapping
Z filter j	2	N/A
Z filter k	5	N/A
Z M	N/A	50
Z K	N/A	250
$\mathcal{A}$ step size	N/A	2.0
$\mathcal{A}$ adapative step factor	N/A	2.0

Table 6.9: Parameters for initial architecture, $\mathcal{A}$, and service selection, Z, searches.

Application	$\mathcal{A}$ Search Budget	Z Search Budget
SAS-15	9,000	20,000
SAS-25	7,200	15,000
SAS-40	10,000	12,000
SAS-65	10,000	7,500

Table 6.10: Budgets for initial architecture and service selection search.

Eleven types of controllers were evaluated in these experiments. The first controller to be evaluated was an autonomic controller using the hill-climbing heuristic pair used in the initial search (see Table 6.9)—this controller will be hereafter referred to as `HC Original` since it employs the original heuristic pair used in the development process. The next eight controllers were autonomic controllers each using one of the heuristic pairs developed with the meta-optimization process described in Section 4.7 on a separate 30 component SAS.The tenth controller is an `Overall Best` meta-controller employing the eight meta-optimized heuristic pairs. The eleventh controller is a meta-controller, `Random MC`, that randomly selects one of the eight meta-optimized heuristic pairs whenever it needs to conduct a re-architecting search. Finally, an uncontrolled system is tested and will serve as the control

for the experiments.

The eight meta-optimized heuristic pairs were formed from four architecture, $\mathcal{A}$, search algorithms and two service selection, Z, search algorithms. The four $\mathcal{A}$ search algorithms were:

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 (with no adaptive step), 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).

The two Z search algorithms were:

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 form the following heuristic search pairs: 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 eleven controllers and the control were evaluated on the four SASes, thus yielding 48 experiments. Each experiment was replicated 100 times with different seeds for the pseudo-random number generator (PRNG) streams controlling the simulation and the behavior of

Application	$\mathcal{A}$ Search Budget	Z Search Budget
SAS-15	128	1,537
SAS-25	100	1,200
SAS-40	80	960
SAS-65	63	756

Table 6.11: Re-architecting budgets for architecture, $\mathcal{A}$, and service selection, Z, search.

the autonomic controller. To reduce unnecessary experimental variance, the same 100 seeds for the simulation event PRNG stream were used for each of the controllers and the control.

The budgets for the architecture, $\mathcal{A}$, and service selection, Z, search in these experiments can be found in Table 6.11. These budgets were selected because the search can be completed in near real-time (approximately 6-7 seconds on a system with dual quad-core 2.4 GHz CPUs).

The number of training replications for the `Overall Best` meta-controller was set to 1.

Controller	Lower Bound	Mean	Upper Bound
uncontrolled	0.9089	0.9111	0.9133
hc-original	0.9192	0.9200	0.9209
`HC-HC`	0.9234	0.9240	0.9247
`HC-EP`	0.9230	0.9237	0.9245
`BS-HC`	0.9234	0.9241	0.9248
`BS-EP`	0.9237	0.9244	0.9251
`EP-HC`	0.9235	0.9242	0.9249
`EP-EP`	0.9234	0.9241	0.9248
`SA-HC`	0.9235	0.9243	0.9250
`SA-EP`	0.9234	0.9240	0.9247
`Random MC`	0.9233	0.9240	0.9248
`Overall Best`	0.9232	0.9239	0.9246

Table 6.12: 95% confidence intervals for average U_g on SAS-15.

Table 6.12 shows 95% confidence intervals for mean U_g over the course of the SAS-15 experiments. The box and whiskers plot in Fig. 6.6 shows the distribution of results. The

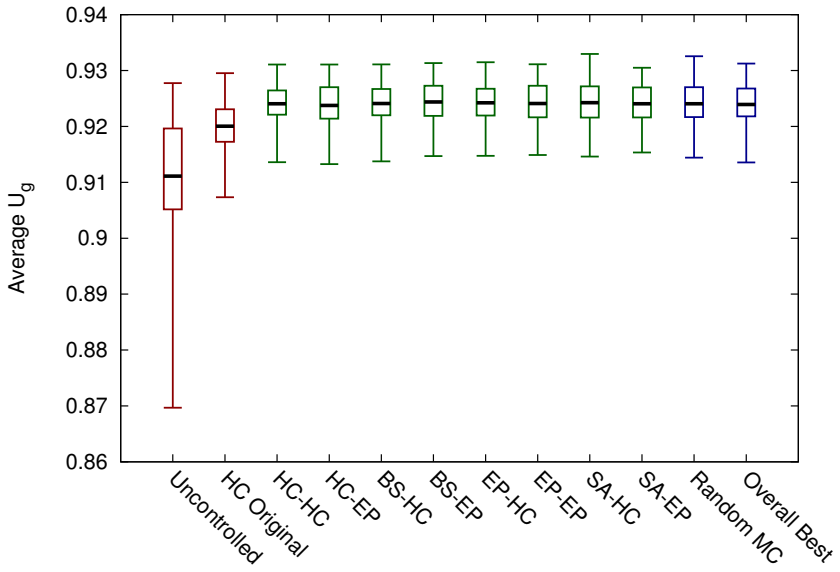


Figure 6.6: Box plot showing results on SAS-15 application.

whiskers show the maximum and minimum average U_g. The boxes show the first and third quartiles, while the black horizontal bar shows the mean.

Applying the Tukey-Kramer procedure to the SAS-15 data demonstrates that all eleven controlled systems performed significantly better than the uncontrolled system at the $\alpha = 0.01$ level. The Tukey-Kramer procedure also shows that all eight meta-optimized heuristic pairs and both meta-controllers were superior to `HC Original` at the $\alpha = 0.01$ level. The Tukey-Kramer procedure yielded no other significant results at the $\alpha = 0.01$, $\alpha = 0.05$, and $\alpha = 0.10$ levels on the SAS-15 experiments.

Table 6.13 shows 95% confidence intervals for mean U_g over the course of the SAS-25 experiments. The box and whiskers plot in Fig. 6.7 shows the distribution of results.

The most striking feature of the SAS-25 experimental results is the poor performance of `HC Original`. This poor performance is likely due to a negative interaction between the neighborhood filters employed by `HC Original` and the structure of the SSSes associated with the SAS-25 (see Table 6.6). The SSS filter, k, is set to two in `HC Original` for both architecture, $\mathcal{A}$, search and service selection, Z, search. The heuristic search algorithm likely finds itself stuck in dead-end neighborhoods when the two worst performing SSSes

Controller	Lower Bound	Mean	Upper Bound
uncontrolled	0.6063	0.6082	0.6102
hc-original	0.5602	0.5638	0.5673
HC-HC	0.6382	0.6387	0.6392
HC-EP	0.6397	0.6401	0.6406
BS-HC	0.6419	0.6422	0.6426
BS-EP	0.6423	0.6427	0.6431
EP-HC	0.6395	0.6399	0.6402
EP-EP	0.6398	0.6402	0.6406
SA-HC	0.6402	0.6406	0.6409
SA-EP	0.6408	0.6411	0.6415
Random MC	0.6411	0.6414	0.6418
Overall Best	0.6419	0.6423	0.6426

Table 6.13: 95% confidence intervals for average U_g on SAS-25.

have a security option metric. Neighbors are generated with upgraded security levels but the resulting increases in execution time and reductions in thoughput may more than offset any improvement in security utility. The HC-HC heuristic algorithm performs much better—the SSS filter in HC-HC's architecture search is set to five, and the service selection search does not use a filter.

The Tukey-Kramer procedure was applied to assess the statistical significance of the differences. HC-Original was significantly worse than all other controlled systems and the uncontrolled system at the $\alpha = 0.01$ level. All other controlled systems were superior to the uncontrolled system at the $\alpha = 0.01$ level. Overall Best, BS-EP, and BS-HC provided superior performance to HC-HC at the $\alpha = 0.01$ level. BS-EP was also better than EP-HC at the $\alpha = 0.1$ level. The Random MC was superior to HC-HC at the $\alpha = 0.1$ level.

Table 6.14 shows 95% confidence intervals for mean U_g over the course of the SAS-40 experiments. The box and whiskers plot in Fig. 6.8 shows the distribution of results.

These results show the importance of the service selection search algorithm on the SAS-40. On the SAS-40, controllers using evolutionary programming for the service selection search generally perform much better than controllers using hill-climbing.

Using the Tukey-Kramer procedure again, all controlled systems provided superior performance over the uncontrolled system at the $\alpha = 0.01$ level. Overall Best, BS-EP, and

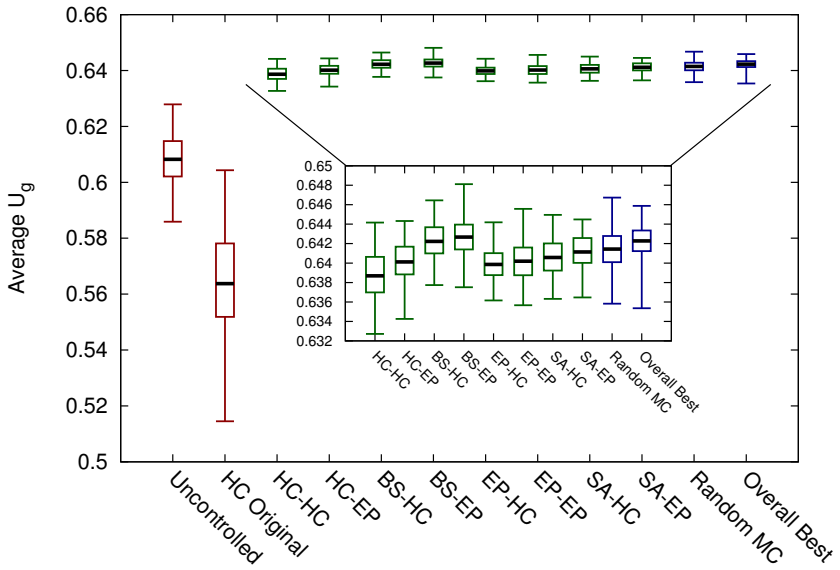


Figure 6.7: Box plot showing results on SAS-25 application.

`HC-EP` were superior to `SA-EP`, `HC-HC`, `EP-HC`, `SA-HC`, and `HC Original` at the $\alpha = 0.01$ level. Additionally, `Overall Best` and `BS-EP` were superior to `BS-HC` and `HC-HC` at the $\alpha = 0.01$ level; `BS-EP` was also superior to `Random MC` at the $\alpha = 0.1$ level. `HC-EP` was better than `BS-HC` at the $\alpha = 0.05$ level.

Other SAS-40 Tukey-Kramer results showed that `Random MC`, `EP-EP`, `BS-HC`, `SA-EP`, `HC-HC`, and `EP-HC` performed better than `SA-HC` and `HC Original` at the $\alpha = 0.01$ level. `EP-EP` was also better than `EP-HC` at the $\alpha = 0.01$ level and `HC-HC` at the $\alpha = 0.1$ level. `Random MC` was superior to `EP-HC` at the $\alpha = 0.1$ level. Finally `HC Original` outperformed `SA-HC` at the $\alpha = 0.01$ level.

Table 6.15 shows 95% confidence intervals for mean U_g over the course of the SAS-65 experiments. The box and whiskers plot in Fig. 6.9 shows the distribution of results.

The experimental results for SAS-65 again show a clear difference between controllers that use evolutionary programming for service selection search vice hill-climbing. Controllers using more exploratory architecture search algorithms (e.g., evolutionary programming and simulated annealing) also struggled.

With the application of the Tukey-Kramer procedure, all controllers except `EP-HC` and

Controller	Lower Bound	Mean	Upper Bound
uncontrolled	0.8349	0.8591	0.8832
hc-original	0.9035	0.9048	0.9061
HC-HC	0.9292	0.9298	0.9304
HC-EP	0.9510	0.9517	0.9524
BS-HC	0.9341	0.9346	0.9351
BS-EP	0.9558	0.9563	0.9568
EP-HC	0.9239	0.9246	0.9253
EP-EP	0.9458	0.9462	0.9466
SA-HC	0.9035	0.9045	0.9055
SA-EP	0.9319	0.9326	0.9334
Random MC	0.9398	0.9405	0.9412
Overall Best	0.9537	0.9543	0.9550

Table 6.14: 95% confidence intervals for average U_g on SAS-40.

SA-HC were found to be superior to the uncontrolled system at the $\alpha = 0.01$ level. Overall Best, BS-EP, and HC-EP provided superior performance over all other controllers at the $\alpha = 0.01$ level.

Other SAS-65 Tukey-Kramer results showed that Random MC, EP-EP, and SA-EP were superior to BS-HC, HC-HC, EP-HC, SA-HC, and HC Original at the $\alpha = 0.01$ level. EP-EP was additionally superior to SA-EP at the $\alpha = 0.05$ level. BS-HC and HC-HC provided superior performance over EP-HC, SA-HC, and HC Original at the $\alpha = 0.01$ level. HC Original was better than EP-HC and SA-HC at the $\alpha = 0.01$ level. Finally, the uncontrolled system and EP-HC were superior to SA-HC at the $\alpha = 0.05$ level.

6.5 Meta-Optimization Experiments

This section describes experiments using the heuristic pairs generated by the meta-optimization process on the SAS-65 in Section 4.7. As described in Section 4.7, twelve finalist problems (labeled A through L) were selected from thousands of collected sample problems. A meta-optimization procedure was applied to each of the finalist problems, which produced twelve corresponding meta-optimized heuristic pairs. (Note that these meta-optimized heuristic pairs are distinct from those described in Section 6.4 which were

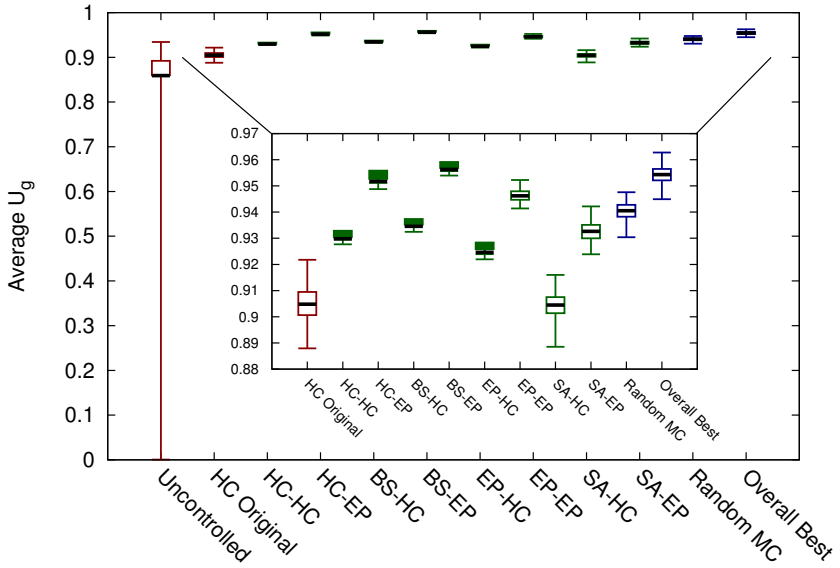


Figure 6.8: Box plot showing results on SAS-40 application.

meta-optimized on a different 30 component SAS.However, the `BS-EP` heuristic pair will be retained and used as a control here.) This work was reported in [38].

The meta-optimization experiments were conducted on the SAS-65.

Figure 6.10 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 first and third population quartiles, the black bar shows the mean, and the whiskers show the maximum and minimum.

The average U_g maintained over the 100 simulations with 95% confidence intervals is presented in Table 6.16. 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 the statistical significance of the results is assessed.

The Tukey-Kramer procedure was applied 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

Controller	Lower Bound	Mean	Upper Bound
uncontrolled	0.8010	0.8036	0.8062
hc-original	0.8077	0.8101	0.8126
HC-HC	0.8192	0.8203	0.8213
HC-EP	0.8512	0.8519	0.8527
BS-HC	0.8222	0.8232	0.8243
BS-EP	0.8520	0.8527	0.8535
EP-HC	0.8020	0.8032	0.8044
EP-EP	0.8385	0.8396	0.8407
SA-HC	0.7983	0.7997	0.8011
SA-EP	0.8348	0.8360	0.8373
Random MC	0.8361	0.8373	0.8386
Overall Best	0.8492	0.8501	0.8510

Table 6.15: 95% confidence intervals for average U_g on SAS-65.

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, the extra variance caused by the inferior performance of certain heuristic pairs was removed 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, the following observations are made:

- 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.

After further reducing the variance to permit comparisons among the top three heuristic pairs (problem K, for problem L, and the control), the following was observed:

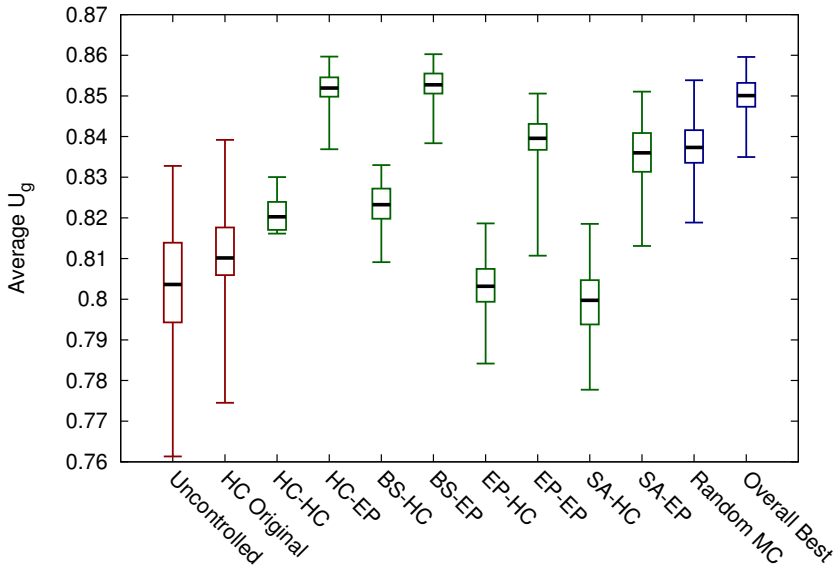


Figure 6.9: Box plot showing results on SAS-65 application.

- 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.

6.6 Context-Best Meta-Controller Experiments

Experimental results for the three different `Context-Best` meta-controllers are presented in this section: a KNN meta-controller (`KNN MC`), an offline SVM meta-controller (`Offline SVM`), and an online SVM meta-controller (`Online SVM`).

6.6.1 KNN Meta-Controller Experiments

A `Context Best` meta-controller using KNN, termed the `KNN MC`, was compared to the `Overall Best` meta-controller and the `Random MC` meta-controller on the SAS-25 (this work was originally presented in [39]). The number of training replications for both meta-controllers was set to 1 in these experiments.

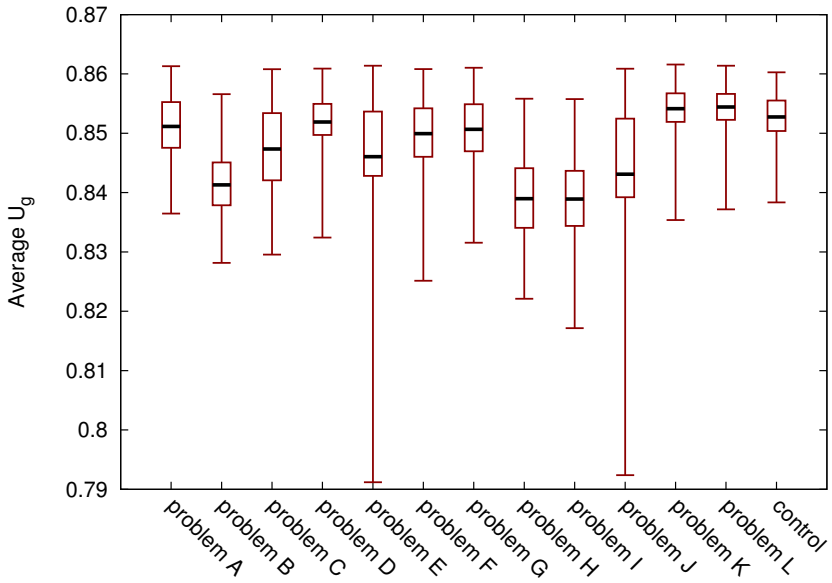


Figure 6.10: Box plot showing simulation results for meta-optimized heuristic pairs on SAS-65.

As in the scalability and meta-optimization experiments, each experiment was replicated 100 times. However, these experiments were conducted with an older version of the simulation software, and the PRNG stream controlling the events was not preserved. This resulted in higher experimental variance for these experiments compared to others discussed in this chapter.

Figure 6.11 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 the statistical significance of the results is assessed.

Table 6.17 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 system near the starting U_g most of the time. Occasionally, a critical SP will fail, and it is either not possible or very difficult to maintain the current U_g. The overall duration of failure events

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

Table 6.16: 95% confidence intervals for average U_g on meta-optimized heuristic pairs.

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 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
`Overall Best`	0.64228	0.64263	0.64297
`KNN MC`	0.64129	0.64164	0.64200
`Random MC`	0.64081	0.64119	0.64157

Table 6.17: 95% confidence intervals for net overall average U_g.

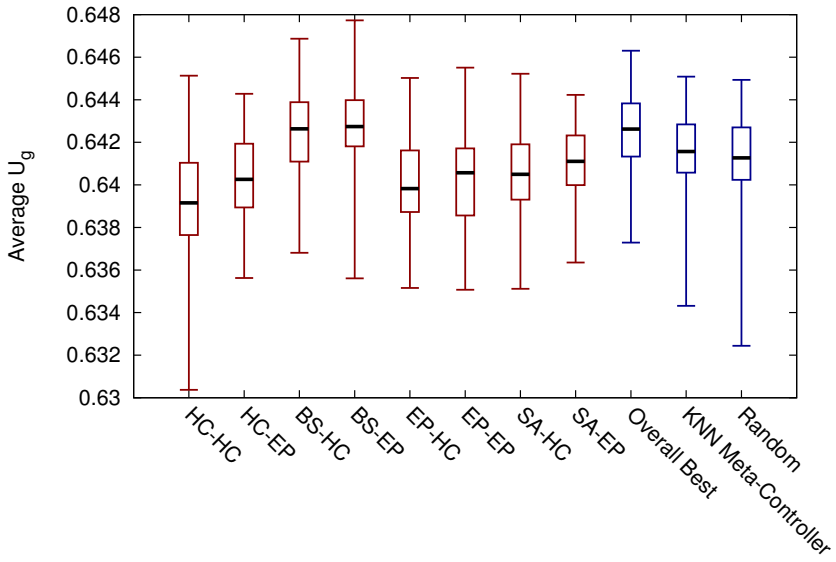


Figure 6.11: Box plot showing the quartiles of experiments with `KNN MC`.

The Tukey-Kramer procedure was applied to perform a simultaneous pair-wise comparison of the eleven controllers (eight simple controllers and three meta-controllers) in Table 6.17. It was determined that `Overall Best` was significantly better than `Random MC` 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`). The Tukey-Kramer procedure was further applied on just the results of the three meta-controllers, and it can be concluded that `Overall Best` was significantly better than the `KNN MC` and that the `KNN MC` was significantly better than `Random MC` at the 95% confidence level.

Figure 6.12 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.

Table 6.18 shows the heuristic pair performance data collected by the `Overall Best`

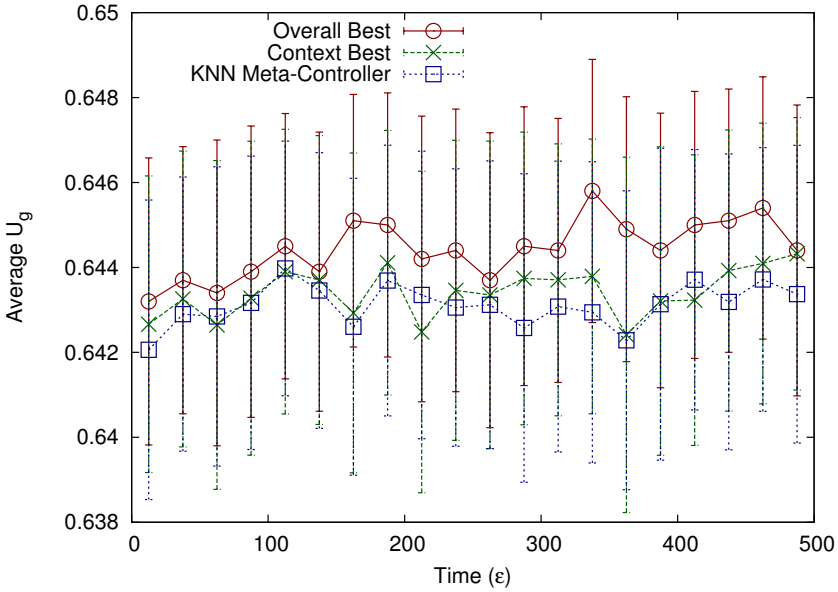


Figure 6.12: The average U_g over time with 95% error bars in `KNN MC` experiments.

meta-controller. Heuristic pairs employing hill-climbing for service selection search perform poorly in comparison to heuristic combinations employing evolutionary programming for service selection search. The best heuristic pair is `BS-EP`, using beam search in the architecture search and evolutionary programming in the service selection search.

Heuristic Search Pair	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 6.18: Performance tables collected by `Overall Best` with 95% confidence intervals.

The evolving behavior of the meta-controllers can be seen in Fig. 6.13. The data series labeled *early* in Fig. 6.13 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. By the second half of the simulation, `Overall Best` has clearly converged to the most overall effective heuristic pair, `BS-EP`. From the data presented in Fig. 6.13, it is unclear if the `KNN`

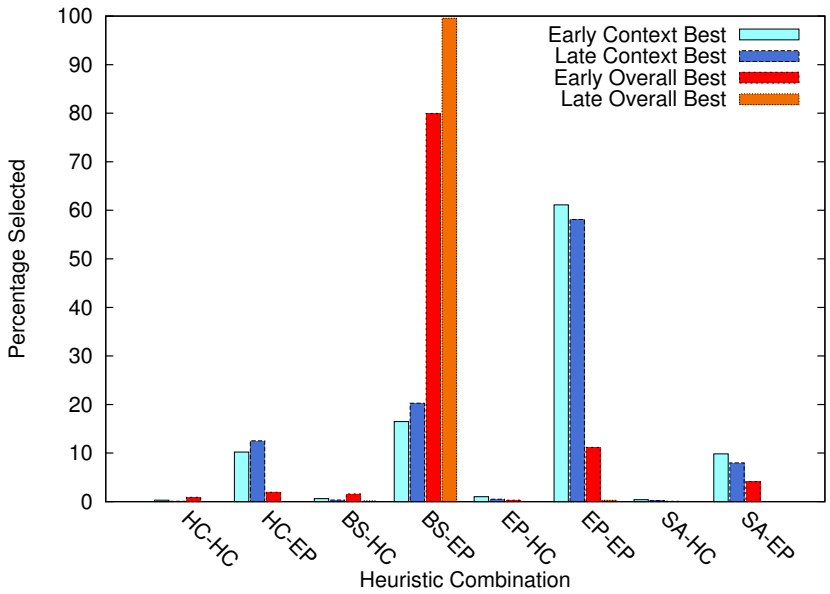


Figure 6.13: Percentage of time a heuristic pair was selected by the meta-controllers.

`MC` is converging. Using paired observations, it was determined that the small increase in heuristic pair `BS-EP` being selected is statistically significant (2.08% $\pm$ 1.97%) at the 95% confidence level.

To better understand the differences in heuristic pair selection between the `KNN MC` and the `Overall Best` meta-controllers, 2,000 re-architecting problems encountered in the simulations were collected. Each heuristic pair was tested against each problem 30 times. For each problem, the average U_g found by all the heuristic pairs was calculated. Then, the relative performance of each heuristic pair on each problem was determined.

Similar to plots presented in Section 5.3.5, Fig. 6.14 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. 6.14 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. 6.14, 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.

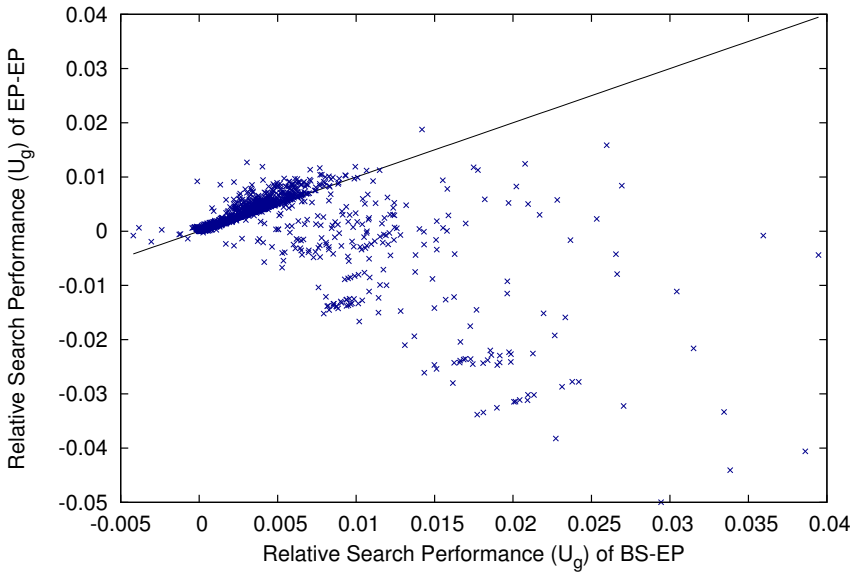


Figure 6.14: Scatter plot of relative heuristic pair performance on 1,935 re-architecting problems.

The KNN algorithm used in the KNN MC is not capable of accounting for the risk that picking incorrectly EP-EP over BS-EP could lead to a relatively large drop in performance. The KNN MC meta-controller would need to correctly identify the best heuristic pair about 85% of the time to equal the performance of Overall Best. It is unlikely that 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 pair for each problem, and
- a relatively large number of fields in the problem characterization.

6.6.2 Offline SVM Meta-Controller Experiments

An offline SVM meta-controller, `Offline SVM`, was developed using the procedure described in Section 5.3.5. The meta-controller used the twelve heuristic pairs (labeled A through L) produced by the meta-optimization procedure described in Section 4.7. An autonomic controller operating with heuristic pair `BS-EP` collected 1,040 sample problems on the SAS-65. The sample problems were tested against each of the twelve heuristic pairs (A through L) 50 times each.

Controller	Lower Bound	Mean	Upper Bound
uncontrolled	0.8010	0.8036	0.8062
E	0.8436	0.8461	0.8485
G	0.8496	0.8507	0.8518
L	0.8537	0.8544	0.8552
`Offline SVM`	0.8487	0.8506	0.8525

Table 6.19: 95% confidence intervals for mean U_g on SAS-65 with `Offline SVM`.

The heuristic pair combination yielding the greatest `AggregateScore` was heuristic pair E and heuristic pair G. The `Offline SVM` was trained with heuristic pair E and heuristic pair G. The trained `Offline SVM` was tested on the SAS-65 and the results are shown in Table 6.19. Results from uncontrolled, E, G, and L are reprinted here for convenience of comparison—heuristic pair L provided the best observed performance on SAS-65 in Section 6.5. A different view of the results is provided in Fig. 6.15.

Applying the Tukey-Kramer procedure to autonomic controllers using E, G, and L as well as the `Offline SVM`, no statistically significant difference were found between `Offline SVM` and the three autonomic controllers at the $\alpha = 0.1$ level. These results show that on the SAS-65 application, the `Offline SVM` was unable to improve upon autonomic controllers

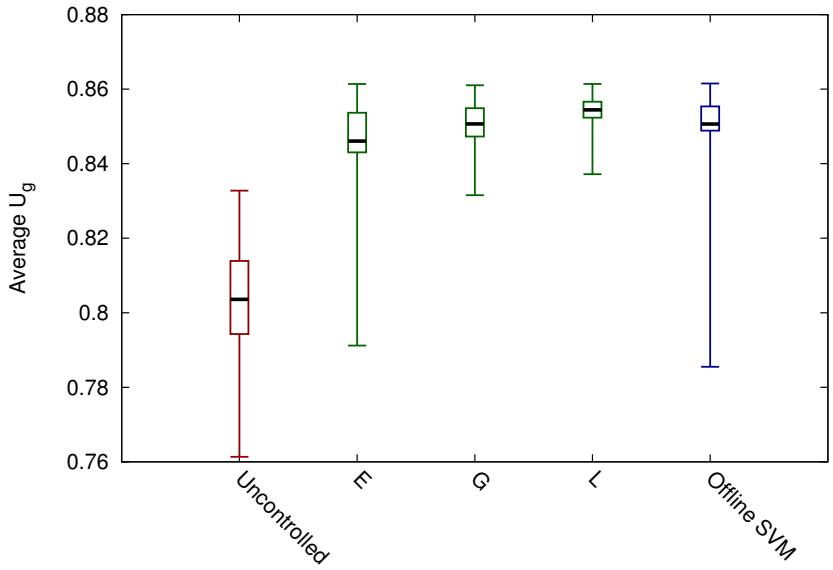


Figure 6.15: Box plot showing results of `Offline SVM` on SAS-65 application.

using the meta-optimized heuristic pairs.

6.6.3 Online SVM Meta-Controller Experiments

An online SVM meta-controller, `Online SVM` was tested on the SAS-40 using the meta-optimized heuristic pairs described in Section 6.4. The results presented here are based on the first 96 experimental replications completed (the last 4 replications are being conducted at the time of this writing).

The majority of the `Online SVM` meta-controllers initially used the heuristic pair combination `BS-EP` and `EP-EP`. A substantial minority of the `Online SVM` initially used `HC-EP` and `EP-EP` heuristic pair combination. All of the meta-controllers converged to `BS-EP` and `EP-EP` by about 200 ϵ (40% of the simulation duration). The `BS-EP` and the `EP-EP` are complementary heuristic pairs as the beam search here is an exploitative algorithm and the evolutionary programming is an exploratory algorithm.

Confidence intervals for the mean U_g can be found in Table 6.20. The confidence intervals for `Online SVM` and `Overall Best` are nearly separated. Applying Tukey-Kramer shows that `Online SVM` is superior to `Random MC` at the $\alpha = 0.01$ level and `Overall Best` at the

$\alpha = 0.05$ level. The distribution of these results can be seen in the box plot shown in Fig. 6.16.

Controller	Lower Bound	Mean	Upper Bound
`BS-EP`	0.9558	0.9563	0.9569
`Online SVM`	0.9548	0.9554	0.9559
`Overall Best`	0.9537	0.9543	0.9550
`Random MC`	0.9398	0.9405	0.9412

Table 6.20: 95% confidence intervals for average U_g on SAS-40 with `Online SVM`.

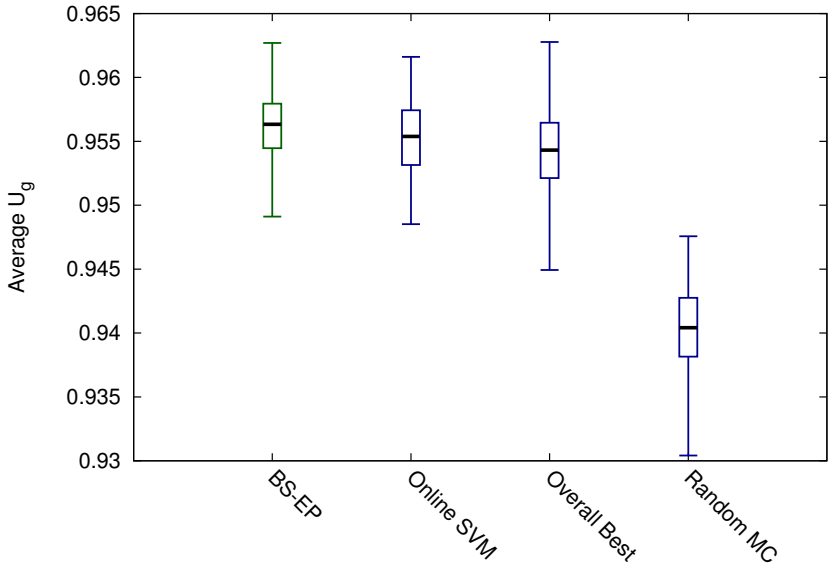


Figure 6.16: Box plot showing results of `Online SVM` on SAS-40.

Figure 6.17 shows the measured global utility over time during the simulation. Though not of statistical significance, it is interesting to note that the `Online SVM` improves over time relative to the other controllers.

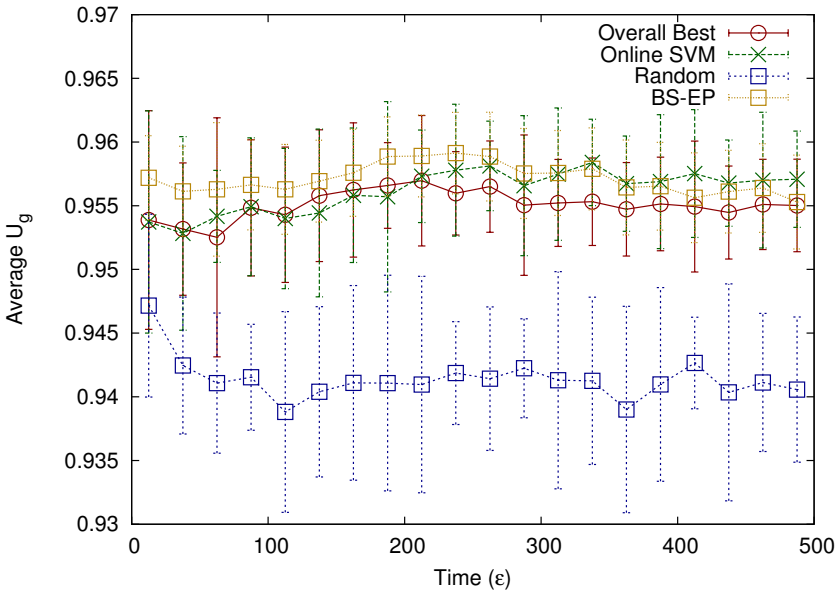


Figure 6.17: `Online SVM` performance over time on SAS-40.

Figure 6.18 shows the distribution of results if the simulation is broken into two halves chronlogically (i.e., the first half covers results in the ϵ range $[1 : 250]$, while the second half covers results in the ϵ range $[251 : 500]$). The autonomic controller using `BS-EP`, the `Overall Best` meta-controller, and the `Random MC` show no significant improvement from the first half of the simulation to the second half. The `Online SVM` meta-controller shows an improvement in the second half of the simulation with 99% confidence (this was found by applying the Tukey-Kramer procedure). This provides an indication that the `Online SVM` meta-controller is learning as the simulation progresses and successfully applying the acquired knowledge to the selection of heuristic search pairs.

Each of the 96 replications for the `Online SVM` meta-controller trained at least 46 SVM models over the course of the simulation. Figures 6.19 and 6.20 examine the average search trajectory for the SVM parameters, C and γ, needed to build the SVM models (see Sections 5.3.5 and 5.3.6 for more information). Figure 6.19 shows the average search for the first 23 SVM models trained, while Fig. 6.20 shows the average search for SVM models 24 through 46. The higher cross-validation rate in Fig. 6.20 is likely due to larger training sets.

It is worth noting the GA population steadily improves as the search continues. This

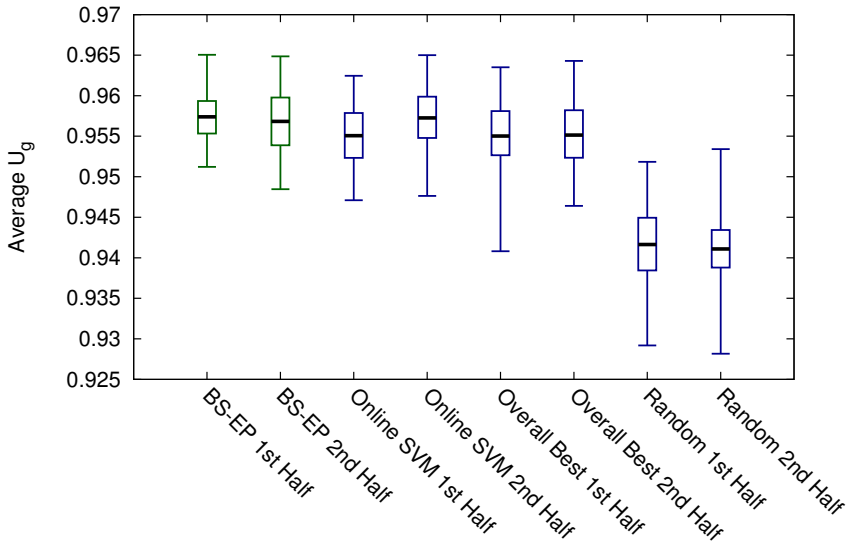


Figure 6.18: Box plot showing 1st and 2nd half simulation results for `Online SVM` on SAS-40.

indicates that the search procedure in Section 5.3.5 is following signal vice noise, which was a concern due to the self-similar properties observed on the cross-validation topologies.

Figure 6.21 shows the weighted prediction accuracy of the `Online SVM` meta-controller's heuristic pair selections. Linear regression was performed on this scatter plot (the correlation, r, was 0.54). This figure indicates that the accuracy improves as the size of the training set increases.

6.7 Summary of Experimental Results

In all of the scalability experiments, the `Overall Best` meta-controller was one of the top performers. Analysis in 6.6.1 demonstrated that the `Overall Best` meta-controller was able to consistently identify the best available heuristic pair. These experiments demonstrate that the `Overall Best` meta-controller provides good performance with low training overhead.

The scalability experiments demonstrated that most of the autonomic controllers scale well. The scalability experiments also revealed that applying an autonomic controller

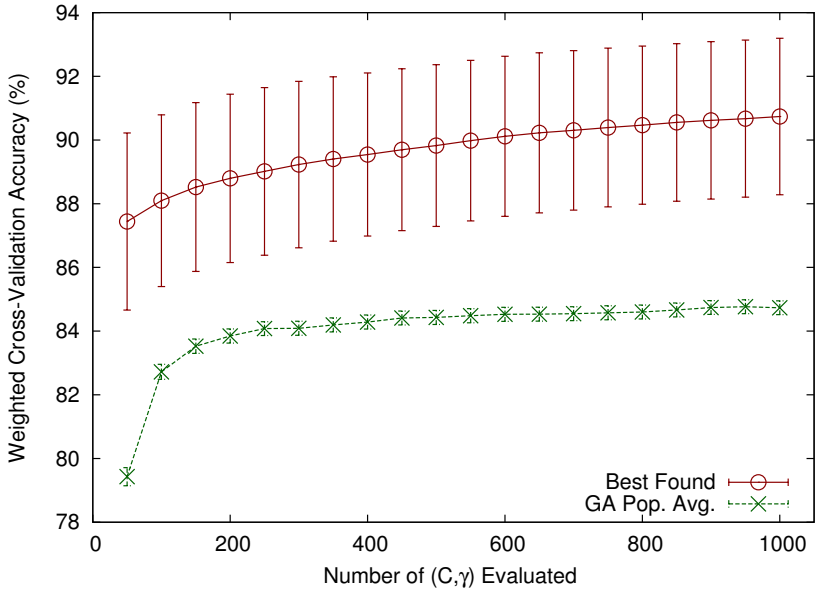


Figure 6.19: SVM parameter search trajectory with 95% CI for SVM models 1-23.

does not necessarily improve the performance. In fact the non-optimized algorithm, `HC Original`, hurt the performance of the system substantially on the SAS-25. On the SAS-65, some algorithms that had been meta-optimized on the development SAS also struggled siginificantly. Autonomic controllers employing the heuristic pair `BS-EP` were notable for their success across multiple levels of scale.

The meta-optimization technique described in Section 4.7 successfully produced two heuristic pairs that were superior to `BS-EP`, the best known heuristic pair from the scalability testing. The results do show that is necessary to test the resulting heuristic pairs to determine which of them provides the best overall performance.

The `KNN MC` struggled due to the inability of the KNN algorithm to account for assymetric risks in mis-classifying optimization problems. The `Offline SVM` meta-controller also struggled to exploit the new heuristic pairs discovered in the meta-optimization process. Further investigation is required for the source of the problems. A hypothesis that explains the poor performance of the `Offline SVM` meta-controller is the following: the offline training set is insufficiently representative of the optimization problem space encountered by the `Offline SVM` meta-controller. Essentially, the SVM hyperplane may extrapolate poorly if the SVM meta-controller wanders into an optimization problem space that is some distance

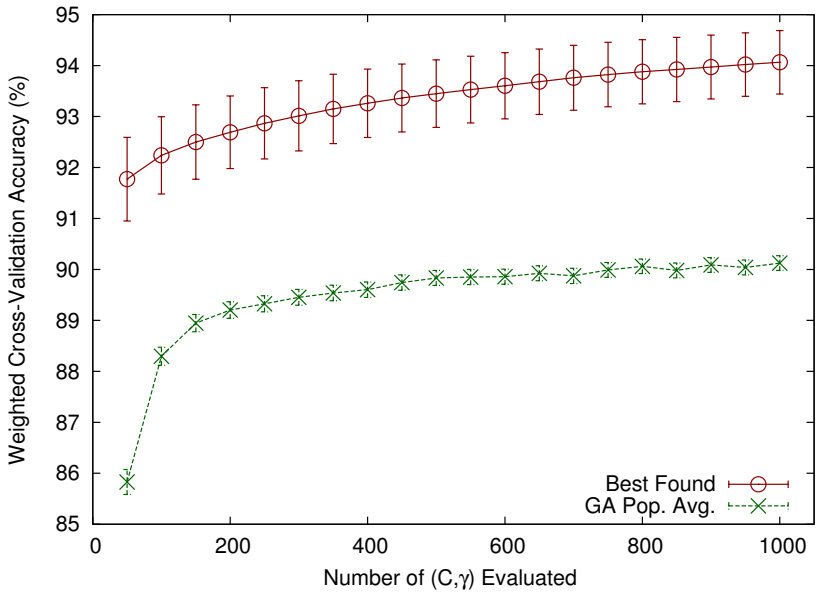


Figure 6.20: SVM parameter search trajectory with 95% CI for SVM models 24-46.

from the problems in the original training set. This results in poor accuracy in predicting the best heuristic pair for the currently faced optimization problem.

Conversely, the results show that the `Online SVM` meta-controller performs better than the `Overall Best` on the SAS-40 system. The `Online SVM` meta-controller has the advantage that its training set is developed from the path in the optimization problem space that it has travelled. As a result, it is more likely that newly encountered problems will be close to the `Online SVM` meta-controller's training set. More testing on other SASes will reveal if the `Online SVM` meta-controller can provide enough performance improvement to justify the extra overhead of training and building SVM classification models.

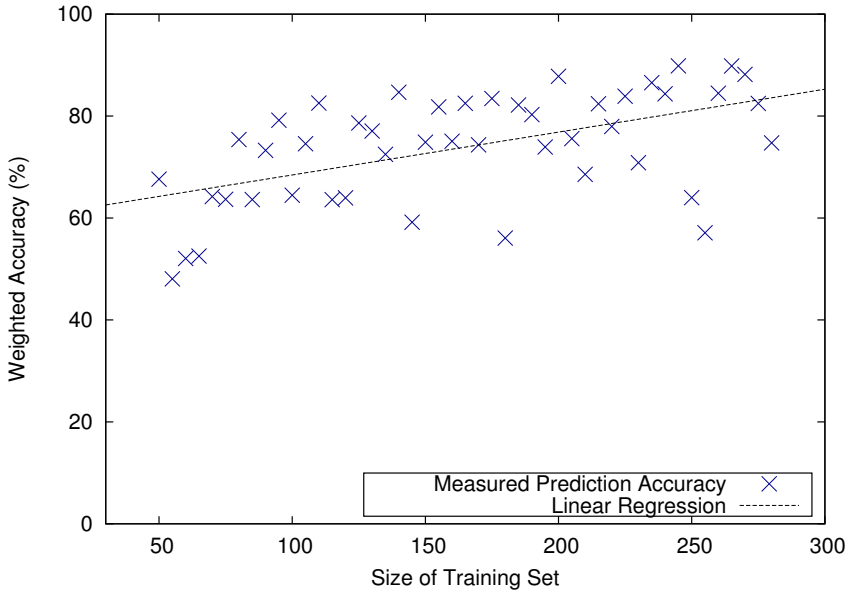


Figure 6.21: `Online SVM` measured prediction accuracy vs size of the training set.

Chapter 7: Concluding Remarks

This chapter begins by reviewing the thesis statement and the criteria defined for its acceptance. The second section examines the contributions made in this work. The third section discusses the management of meta-controllers. The last section looks at promising areas for future investigation.

7.1 Consideration of the Thesis Statement

In this section, a decision will be taken to accept or reject the thesis statement stated in Section 1.4 based on the success criteria defined in Section 1.5. The thesis statement was divided into three statements each with their own criterion for acceptance.

7.1.1 Evaluating Thesis Statement H1

The first statement, H1, of the thesis is reprinted below:

> Autonomic computing techniques can be used to automatically design the architecture of SOA-based software systems and select service providers in a scalable way that optimizes utility.

The corresponding criterion for accepting H1 follows:

> Develop techniques for the automatic design of architectures and corresponding service selection and demonstrate through rigorous experimentation that the methods achieve optimized solutions in a scalable way.

Multiple heuristic search techniques were developed and applied to the automatic design of architectures and service selection. These techniques, supporting technologies, and frameworks are described in Sections 4.3, 4.4, and 4.5. Experimental evidence in Section 6.3.4 demonstrated the success of these techniques.

7.1.2 Evaluating Thesis Statement H2

The second statement, H2, of the thesis is:

> Scalable autonomic computing techniques that self-adapt to new architectures and new service provider selections can be used to maintain optimized levels of utility for SOA-based software systems in the face of failures and performance degradations of service providers.

The criterion for accepting H2 is:

> Develop techniques for the automatic re-architecting and corresponding service selection for SOA-based systems in the face of failures and performance degradation. Demonstrate through rigorous experimentation that the methods maintain high utility solutions in a scalable way.

The re-architecting framework described in Section 4.4 provides a methodology for maintaining system performance as failure and performance degradation events occur. This framework is supported by modeling technologies described in Section 4.3 and utilizes the heuristic search techniques described in Section 4.5. A number of experiments described in Chapter 6 demonstrated the effectiveness of the framework. A specific set of scalability experiments in Section 6.4 showed that the approach from Section 4.4 and that the algorithms detailed in Section 4.5 scale with the size of the managed application.

7.1.3 Evaluating Thesis Statement H3

The third statement, H3, in the thesis is printed below:

> An autonomic meta-controller employing machine learning techniques can measure and tune the autonomic controller's performance, and thus reduce the human effort required to manage the system.

The criterion for accepting H3 is:

> Develop a meta-heuristic agent that adaptively selects heuristics and their parameters to solve self-architecting and service selection optimization problems.

Four different meta-controllers of varying levels of complexity were developed and are described in Chapter 5. All of the meta-controllers are capable of measuring and tuning an autonomic controller.

Experiments in Section 6.4 and Section 6.6.1 revealed that the `Overall Best` meta-controller is capable of identifying the best overall heuristic pair with low overhead. Chapter 6 provided evidence that the `Overall Best` meta-controller was competitive with autonomic controllers that had preselected the best available heuristic pair.

Though `KNN MC` and the `Offline SVM` meta-controllers proved less effective in experiments from Sections 6.6.1 and 6.6.2, early results for the `Online SVM` meta-controller appear promising.

Further, a meta-optimization technique was described in Section 4.7 that optimizes heuristic pairs for specific SASSY applications. The resulting twelve heuristic pairs were tested in experiments described in Section 6.5, and two of the generated pairs exceeded the performance of the previous best performing heuristic pair.

The criteria for the three thesis statements have been met. The thesis statement is accepted.

7.2 Reviewing the Contributions

This dissertation makes a number of contributions in autonomic frameworks, performance modeling, and run-time adapation of autonomic systems.

7.2.1 Frameworks

Three centralized autonomic frameworks were presented:

1. an autonomic controller framework for managing SOA applications,
2. a meta-optimization framework for improving heuristic search algorithms, and
3. a meta-controller framework for the run-time adaptation of the autonomic controller.

The autonomic controller provided by the first framework fulfills the Goal Management Layer function (as described in Section 2.5.7) in the SASSY project. Both the meta-optimization framework and the meta-controller framework enhanced the performance of

the autonomic controller. In addition to enhancing the autonomic controller, these two frameworks could also be adapted to other autonomic computing problem domains beyond SOA application management.

7.2.2 Performance Models

The autonomic controller has the ability to automatically generate performance models for specific SOA applications. The techniques for producing these performance models may be useful to other SOA monitoring and management systems.

7.2.3 Run-time Adaptation of Autonomic Controllers

Autonomic controllers were adapted at run-time by meta-controllers. Four different meta-controller techniques have been described here in Chapter 5. The concept of the meta-controller could be broadly applied across autonomic computing and robotics.

7.2.4 Experimental Examination

The frameworks and technologies developed for this dissertation have undergone rigorous experimental evaluation. This experimental evaluation included scalability testing. The experimental simulation techniques described here could be used to evaluate other SOA systems.

7.2.5 Minor Contributions

A number of minor contributions have been made as well. They are detailed below.

Population Overlap Metric

In assessing relative behavior of heuristic pair combinations, a general method for measuring the overlap of two populations of points was developed (see Section 5.3.5). This method could be extended with the k-means clustring algorithm [8]. This would allow the two populations to be broken up into clusters before applying the overlap measurement algorithm. Such an algorithm might have many uses in the fields of data mining and pattern recognition.

Approach for SVM Penalty Weights on Asymmetric Data Sets

A reasoned approach to SVM penalty weights for controlling one-sided overfitting of asymmetric data sets was presented in Section 5.3.5. This technique could be valuable to SVM practicioners who work with small, unbalanced training sets.

High Performance Data Structure Templates

The ability of the heuristic search algorithms to rapidly evaluate hundreds of thousands of SASSY systems in a matter of seconds is due to the high performance data structure templates developed in C++. These templates blend the best features of arrays, linked lists, and hash tables. Additionally, these templates provide memory management functionality that speeds performance by massively reducing the number of requests for memory allocation—a relatively slow operation in modern computing systems.

Library of Heuristic Search Templates

A library of multi-threaded heuristic search algorithm templates was developed in C++ using the high performance data structure templates. This heuristic search library is general and can be applied to any C++ object class that can provide a scoring function, a hashing function, a neighborhood generation function, and a mutation function.

7.3 Discussion: Managing the Meta-Controller

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

Although the meta-controller will require some initial effort from human administrators, there is a strong 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-controller. This dynamism can throw an autonomic controller off-kilter (e.g., the trap the `HC Original` autonomic controller fell into on the

SAS-25 in Section 6.4).

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

7.4 Future Work

There are a number of areas for future research to explore.

7.4.1 Additional Heuristic Search Algorithms

There are some interesting heuristic search algorithms that were not implemented in this work that may provide good performance for SASSY meta-controllers. Foremost among these heuristic search algorirthms is Ant-Colony Optimization (ACO) [29]. ACO works well on problems where solutions have interchangeable parts—both architectures and service selections meet this criterion.

7.4.2 Meta-Optimization

There are some improvements that could be made to the meta-optimization approach described in Section 4.7. The meta-optimization could benefit from meta-heuristic algorithms that converge more rapidly than the GA algorithm used here. The process of selecting finalist problems for the meta-optimization is rudimentary and could be better developed. Finally, the fitness function for a heuristic pair derived from its performance on a finalist problem could be modified to use a blend of the heuristic pair's performance on multiple finalist problems.

7.4.3 Meta-Controller

The `Overall Best` meta-controller may benefit from a more adaptive training strategy. For example, the number of training replications could be elevated when the `Overall Best` meta-controller first becomes operational. As the `Overall Best` meta-controller becomes confident that it has converged onto the best heuristic pair, it could wind down its training activity.

Prediction accuracy and risk evaluation are the biggest challenges faced by the three `Context Best` meta-controllers (see Section 6.6). One possibility for improving their performance would be the introduction of a hybrid approach of the `Offline SVM` and the `Online SVM` meta-controllers. The `Offline SVM` meta-controller would bring its knowledge of the overall performance of the heuristic pairs and the costs of mis-classification. The `Online SVM` meta-controller would provide more relevant optimization problems to the constuction of the SVM hyperplane decision boundary. Additionally, the `Online SVM` may also be able to better represent the nearby optimization problem space by studying mutations of recently encountered problems or by studying projected future optimization problems.

Additionally, it may be valuable to revisit the characterization of the optimization problems (see Section 5.3.1). A simpler representation of optimization problems may help the SVM avoid overfitting noise.

Appendix A: High Performance Data Structure Templates

This appendix describes the generalized high performance data structures that were developed to suppport two-level re-architecting search in near-realtime. The first section provides a brief overview of C++ templates. *List* data structures and the abstract `List` template are covered in the second section. The third and fourth sections discuss the templates `ArrayList` and `LinkedList` respectively. The fifth section examines the `HashTable` template, while the sixth section analyzes the `Set` template that blends the features of multiple data structure temples.

A.1 C++ Templates

The C++ programming language contains two powerful abstraction features: class templates and class inheritance. By combining these abstraction features with support for low-level programming, the C++ programming language sets itself apart from other modern programming languages[123].

Class templates are defined in C++ by specifying one or more abstract data types before defining the class. The implementation of class template methods must include the abstract data types in their scope. When a computer programmer wishes to use a class template, the computer programmer must specify a data type to serve for each abstract data type in the template definition. When the computer program is compiled, the compiler will create a new specific class from the template for each unique combination of specified data types.

Thus, classes and methods written as templates can be applied to many data types. This flexibility encourages the re-use of computer programming code and reduces the level of effort needed to develop and maintain computer programming code. The computer programming code developed in support of this dissertation uses the template features in C++. High performance data structures written as C++ templates were utilized throughout the code for storing architecture objects, service selection objects, archiecture components, threads, etc.

A.2 List

A *list* is a data structure that holds a number of elements in a certain order. Various methods can insert, delete, and retrieve elements from the *list*. Some lists support the concept of a *cursor* that points to a specific element in the *list*. The *cursor* can be moved to the next element in the *list* via the method `gotoNext()` or the previous element in the *list* via the method `gotoPrior()`. With other methods, the *cursor* can also be moved to either the beginning or end of the *list* [111]. Often a loop will iterate through the list in a manner similar to a *for* loop iterating through an array.

A *list* can be implemented several different ways. The two most common methods for implementing a *list* are through arrays and linked nodes called *linked lists* [19]. In this dissertation, a `List` template was defined with pure virtual methods. Classes that use pure virtual functions are called abstract classes, and cannot be directly instantiated. With pure virtual methods, no specific implementation is provided—it is up to inheriting child classes to provide an implementation. Abstract classes provide a common interface; a program may have a pointer to a a *list* of integers (i.e., `List<int>*`) but may call `gotoNext()` or `gotoPrior` without needing to know if the *list* is implemented with an array or a linked list.

The abstract template class `List` can be seen in the Unified Modeling Language (UML) class diagram in Fig. A.1. Figure A.1 was developed with the Umbrello UML Modeller [51]. Note that the abstract data type is `LE` which stands for *List Element*. When instantiating an `ArrayList` or a`LinkedList` the programmer must specify what data type will be used for `LE`. In Fig. A.1 the data types in the method arguments appear after a ':'. Data types that are followed by an '&' are passed by reference, while data types that are followed by an '*' are memory pointers to specific object instances. Data types returned by a method follow the close of the arguments and another ":".

The following pure virtual methods are defined by `List`:

- `insert(newElement : LE &)`,
- `remove()`,
- `replace(newElement : LE &)`,

- `append(toBeAdded : List< LE >&)`,
- `clear()`,
- `getCursor() : LE`,
- `getCursorRef() : LE&`,
- `gotoNext() : bool`,
- `gotoPrior() : bool`,
- `gotoBeginning() : bool`, and
- `gotoEnd() : bool`.

For the sake of clarity, the pure virtual functions of `List` are not shown in Figure A.1.

The *list* templates used in this dissertation are significant extensions of the C++ template data structures found in [111]. The implementation for the hash table template and the set template are novel to this dissertation.

A.3 ArrayList

The `ArrayList` template provides an implementation of the abstract `List` template using arrays. *Lists* that use arrays are trivial to implement, and many useful algorithms for sorting are designed to operate on arrays [19]. However, *lists* implemented by arrays do have limitations. Some methods in `ArrayList` are relatively computationally expensive. For example, when inserting in the middle of the `ArrayList`, it is necessary to copy each element after the cursor down one space in the array to make room for the new element. Additionally, the size of the array must be specified when the `ArrayList` is instantiated—this sets a maximum number on the number of elements in the *list*.

Two different implementations of `ArrayList` exist in the dissertation code: `ArrayListPlain` and `ArrayListHash`. The `ArrayListPlain` makes no demands on the specified data type—integers, floating point numbers, and characters can all be used in `ArrayListPlain`. The `ArrayListHash` requires the specified data type to implement the methods

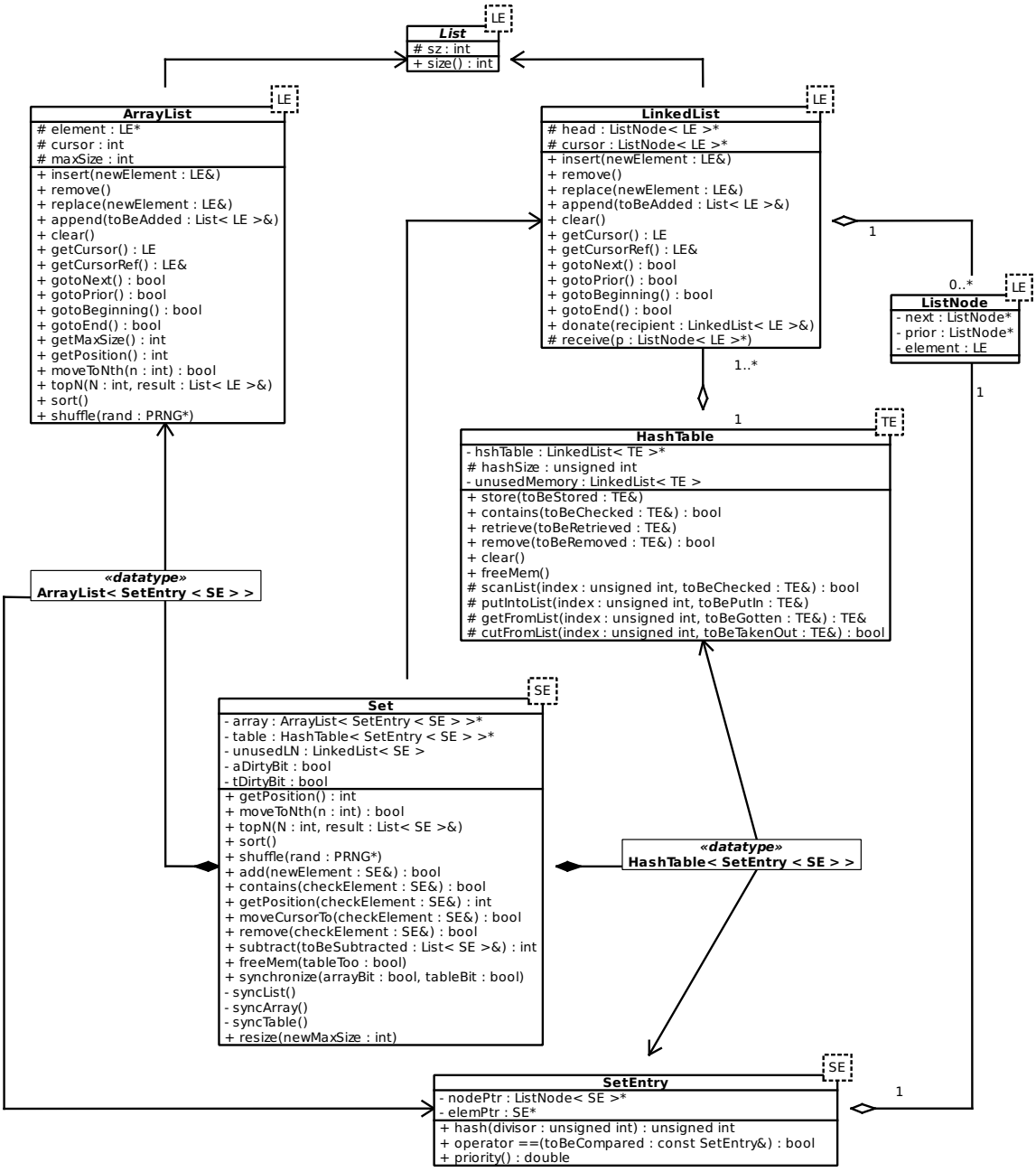


Figure A.1: UML describing the derivation and composition of the `Set` template.

`priority()` and `hash()`—integers, floating point numbers, and characters would require a wrapper class implementing `hash()` and `priority()` to be used in `ArrayListHash`.

The `priority()` method is used for sorting and selecting the top N elements. The `hash()` method helps support the hashing methods for large complex classes such as architectures and service selections that can contain multiple `ArrayList` and `Set` object instances.

A.4 LinkedList

The `LinkedList` implementation of the abstract `List` template uses a series of list nodes. Each list node is a tuple of a data element, a pointer to the next list node, and a pointer to the previous list node. Implementations of linked lists that use two pointers in a list node (like the implementation presented here) are called *doubly linked lists* [111]. (*Singly linked lists* use list nodes that contain no pointer to the previous list node.)

At first glance, linked lists appear to be better suited for inserting and removing data. Inserting an element into the `LinkedList` template requires the memory allocation of a new list node, and the modification of four pointers (two in the newly allocated list node, one in the prior list node, and one in the next list node). However, in modern computing systems, allocating memory has become a relatively expensive operation. The `Set` and `HashTable` templates will address this issue.

Two special methods `Donate` and `Receive()` provided by `ListedList` enable two linked lists to pass list nodes back and forth. This can provide performance savings by reducing the number of memory allocation calls made. When using one linked list, another empty linked list can store allocated list nodes that are no longer in use. These nodes can be donated back as needed.

A.5 HashTable

Lists are effective data structures when the programmer wants to perform operations on every element in the data structure or work on sequences of elements within the data structure. Frequently, it is necessary to ascertain if an element is already present in a

data structure; for this task, *lists* are ill-suited. A more appropriate data structure when searching for elements is a hash table.

A hash table is a large array, often much larger than the number of data elements that will be stored in it. Unlike *lists*, hash tables do not retain any ordering information about the elements. When adding an element to the hash table, a hashing function computes a non-negative integer called a *hash* from the properties of the element; the *hash* should be smaller than the size of the array. The element is then stored in the array position denoted by the *hash* value. When two elements with differing properties produce the same hash value, this is termed a *collision*. The goal of a good hashing function is to avoid *collisions* as much as possible.

There are two major approaches to dealing with hash table *collisions* during insertion of an element. The first approach scans for the next empty slot after the hash of the colliding element; this approach places the colliding element in the first empty slot found. This approach gets complicated quickly, as methods looking for elements must now check all contiguously occupied elements appear at and after the *hash* in the array for equivalence. Removing elements in this approach is complicated as the new empty spaces might prevent the successful lookup of an element placed after the removed element. As the hash table fills up, the performance of this first approach can seriously degrade [19].

A second and more elegant approach is to have each entry in the array correspond to a separate linked list. If a collision occurs, another element is just added at the end of the linked list corresponding to that *hash*. The performance of this approach will also degrade with a large number of collisions, but not as severely as with the first approach [19]. This second approach is adopted in the `HashTable` template used in this dissertation. Like the `ArrayListHash`, the `HashTable` template requires data elements to provide a `hash()` method. When a `HashTable` template object is insantiated, the size of the array of linked lists is set to a prime number slightly larger than the size specified in the `HashTable` constructor method. This prime number will be provided as the `hashDivisor` argument in all `hash(unsigned int hashDivisor)` calls.

The performance of the `HashTable` template is enhanced by saving list nodes in a pool when entries are removed from the table (the `donate()` and `receive()` methods of

LinkedList are used to performn this task). When new elements are added to the table, the HashTable template can check to see if a list node in the ususued pool is availble. The method freeMem() will de-allocate the list nodes in the unused pool.

A thread-safe version, HashTableTS template was also developed. This version uses the mutex data structures and methods made available in the BOOST thread library [128]. The HashTableTS template is used in the multi-threaded heuristic search algorithms presented in Appendix B.

A.6 Set

To permit successful re-architecting of large SOA systems in near-realtime, performance and scalability were prioritized during the development of the two-level search. Neighborhood generation algorithms with filtering proved to be particularly challenging. The first draft of this code was both complicated and cumbersome using a combination of ArrayList, LinkedList, and HashTable data structures. It was clear that a new approach was required to improve performance, scalability, and code maintainability.

The Set template was devised to solve the problem and combines the features of linked lists, array lists, and hash tables in a seamless manner. A Set template data structure can be thought of as a hash table that also cares about the ordering of the objects like a list. The Set template inherits from the LinkedList template. Similar to the HashTable template, the Set template maintains a pool of unused list nodes to improve performance.

A key component of the Set template is the SetEntry which is analogous to a list node, though a SetEntry is typically a much smaller object. A SetEntry is a tuple of element pointer and a list node pointer. The SetEntry provides a hash() method which in turn calls the hash() method of the element to which it points. In a similar manner, the SetEntry provides a priority() method and an equivalence operator. SetEntries are stored two different ways in the Set template: in a ArrayList and in a HashTable. The hash table and the array list are synchronized with the linked list.

The Set template does not keep the linked list, the hash table, and the array list in sync after each operation. Instead, the Set template maintains a separate *dirty-bit flag* for the linked list, the hash table, and the array. A call to the method sort() or shuffle() would

be conducted on the list arry of `SetEntries`; the *dirty-bit flag* for the linked list would then be set to true because its ordering would no longer reflect the true ordering of the `Set` data structure. The next time, an operation is performed on the linked list like an `insert()`, the `Set` template will check the linked list's *dirty-bit flag* and re-sync the linked list with the list array (this is easily accomplished since the set entries contain pointers to the list node in the linked list). Because the `SetEntries` are lightweight objects (e.g. 16 bytes on a 64-bit system), re-syncing the list array and the hash table are relatively quick operations. The`Set` template can combine the speedy lookup performance of the hash table, with the insertion/removal performance of a linked list, and the sorting features of a list array. The `Set` template will almost always provide superior performance over using two different data structures and copying data back and forth between the data structures.

It is important to note that the `Set` data structure will only keep one unique copy of an element. This turns out to be a useful feature in local search neighborhood construction and managing training sets for `Context Best` meta-controllers.

The concept of the *No Free Lunch Theorem* [129] pertains here: care must be taken in considering the sequence of operations performed on the `Set` class. It is possible to choose a sequence of operations that will force re-syncing of a data structure prior to each operation. With such operation sequences, performance will be degraded rather than enhanced.

Because the `Set` template is feature-rich, it also substantially simplifies the code where it is used. Beyond the re-architecting search, the `Set` template was utilized in the management of online training sets for the `Context Best` meta-controllers. This reduced the complexity of the resulting code, which made it easier to identify errors in the automated scale management of the training data.

Appendix B: Heuristic Search Algorithm Implementation

This appendix describes the C++ templates used to implement the multi-threaded heuristic search algorithms used in this dissertation. The first section examines the overall structure of the heuristic search templates. The second section examines the `HeuristicSearchMTE` template that provides the support functions and data structures for multi-threaded search. The third section focuses on multi-threaded templates for hill-climbing, beam search, and simulated annealing. The fourth section considers the templates for evolutionary programming and genetic algorithms.

B.1 Heuristic Search Algorithm Template Structure

The heuristic search templates inherit from an abstract base template called `HeuristicSearch`. Two abstract data types are associated with `HeuristicSearch`: `solution` and `torch`. The `solution` abstract data type is the object to be optimized. The `torch` abstract data type represents memory resident data structures used for specific processing related to the `solution` data type. The `solution` data type is expected to provide the methods `score()` and `hash()`. For local search methods, `solution` should also provide `getNeighborhood()`. Evolutionary programming and genetic algorithms expect `solution` to provide `mutate()`, and genetic algorithms further expect `solution` to provide `crossover()`. To support `mutate()` and `crossover()`, genetic algorithms will sometimes utilize a wrapper class to convert the actual solution class to binary strings; the wrapper class then serves as the `solution` data type.

The `HeuristicSearch` template defines the method `search()` as a pure virtual function (see Section A.2); `search` serves as the main interface for all heuristic search algorithm templates. The common interface simplifies the code wherever multiple types of heuristic search algorithms may be instantiated and applied; this includes the autonomic controller, the meta-controllers, and the `Architecture` class.The `search()` method takes one argument, `startingPoint`, a `solution` that is passed by reference. The best `solution` found during the search will be copied into `startingPoint`. Since `startingPoint` is passed by

reference, the best `solution` will be returned to the caller of the search when the search completes.

The other major function of the `HeuristicSearch` template is to provide inheriting templates with access to a `HeuristicSearchProperties` object (represented as the pointer `myProperties`). `HeuristicSearchProperties` objects contain a large number of `get()` and `set()` methods for all heuristic search algorithm parameters. These parameters may be assigned by reading text files or through the meta-optimization process described in Section 4.7. The relationship between `HeuristicSearchProperties` and `HeuristicSearch` can be seen in Fig. B.1.

B.2 Heuristic Template for Multi-threading

There are two abstract classes that inherit from `HeuristicSearch`: `HeuristicSearchST` and `HeuristicSearchMTE`. The `ST` stands for single-threaded and `MTE` stands for Multi-Threaded Element. The `HeuristicSearchST` was used in the development and testing process. This appendix will focus on the `HeuristicSearchMTE` because it was used in the experiments described in Chapter 6.

The `HeuristicSearchMTE`'s primary objective is to streamline the management and coordination of threads for the heuristic search algorithms. The secondary objective of `HeuristicSearchMTE` is to provide data elements and structures that are common to all of the inheriting heuristic search templates.

The `HeuristicSearchMTE` provides two groups of methods. The first group of methods enables threads to communicate with a heuristic search instance. The second group of methods is comprised of internal procedures common to all the heuristic searches, such as `budgetConsumed()`.

The `HeuristicSearchMTE` contains a large number of data structures for managing the concurrent execution of the threads. Many of these data types are provided by the BOOST thread library [128].

Figure B.1: UML describing the heuristic search templates.

B.2.1 Initializing Threads

When a template class inheriting from `HeuristicSearchMTE` is instantiated, the number of threads to be used is collected from `HeuristicProperties`. The `HeuristicSearchMTE` also determines the search budget; the search budget can be either a time limit or a limit on the number of `solution` evaluations.

If the search budget is a time limit, then the `HeuristicSearchMTE` creates an instance of `TimerThread`. The `TimerThread` instance is provided with a pointers to a semaphore, a condition variable, and the heuristic search instance. The `TimerThread` object is provided to the BOOST thread library, which clones the object and starts a new thread executing in a special method, `operator()()`, provided by the `TimerThread` class. The `operator()()` directs the new thread to call the `RegisterTimer()` method provided by `HeuristicSearch-MTE`, and provides the heuristic search instance with a pointer to the thread's copy of the `TimerThread` instance. The thread still executing `operator()()` is blocked while waiting for the condition variable `running` to become true. `HeuristicSearchMTE` provides an interface to inheriting templates to wake up the timer thread later.

The `HeuristicSearchMTE` then begins the process of creating worker threads that will perform most of the real work in the search. Each worker thread requires two semaphores and two condition variables. After the semaphores and condition variables are created, `HeuristicSearchMTE` instantiates a `HeuristicThread` object. At the time of creation, the `HeuristicThread` object is given the pointers to the semaphores, the condition variables, and the `HeuristicSearchMTE`. Like the `TimerThread` object, each `HeuristicThread` object is provided to the BOOST thread library, which again clones the object and starts a new thread executing in the `HeuristicThread` implementation of `operator()()` (shown in Listing B.1).

B.2.2 Main Worker Thread Loop

A worker thread remains in the loop that starts on line 5 of Listing B.1 until the heuristic search instance is destructed. One of the first actions in the loop taken by a thread is to call `HeuristicSearchMTE`'s method `joinReadyList()` (line 13). The `joinReadyList()` provides a thread-safe method to put the pointer of the `HeuristicThread` into heuristic

search's linked list called `sleepingThreads` (also known as the ready list). After joining the ready list, the thread returns to `HeuristicThread`'s `operator()()` and is blocked while waiting for the condition variable `running` to become true (line 17).

When the method `TaskOneThread()` provided by `HeuristicSearchMTE` is called by the heuristic search, `running` will be set to true for one thread in the `sleepingThreads` list. The selected thread's enumerated parameter `action` is also set by the method `TaskOneThread()`. `TaskOneThread()` then notifies the thread that the value of running has changed via the condition variable. Based on the value of `action`, the selected thread chooses to call one of three possible methods: `generate()` (line 21), `evaluate()` (line 23), and `singleThreadSearch` (line 25).

The interfaces to `generate()`, `evaluate()`, and `singleThreadSearch()` are defined by `HeuristicSearchMTE` as pure virtual functions; these methods are actually implemented by the templates that inherit from `HeuristicSearchMTE`. The `generate()` methods implement a mechanism for the thread to generate new `solutions` in a manner that is coordinated with the other worker threads. The `evaluate()` methods assign the thread a number of `solutions` to evaluate. The `singleThreadSearch()` methods use the selected worker thread to perform all of the search functions by itself. When the selected thread completes the method call, it returns to the beginning of the while loop on line 5 and repeats the process of joining the ready list and waiting to run again.

Heuristic search threads sometimes need to signal the threads to stop their work and either return to the ready list or prepare for the destruction of the heuristic search object. If the thread receives an interrupt between lines 8 and 28, the point of execution moves to the exception handler starting on line 30. If the thread was running, it rejoins the ready list on line 35. Using the interrupt semaphore, `intrrptMutex`, and the interrupt condition variable, `intrrptCV`, the thread notifies the heuristic search that it has received and processed the interrupt (line 40).

Disabling interruption on line 9 is crucial—if an interrupt is received after joining the ready list on line 13 but before `running` is set to false on line 14, the ready list will make a duplicate entry for this thread on line 35. The condition of duplicate entries in the ready list would eventually lead to a failure.

```
1  template <class solution, class torch>
2  void HeuristicThread<solution,torch>::operator () ()
3  {
4    interrupted = false;
5    while(true)
6    {
7      try
8      {
9        boost::this_thread::disable_interruption di;
10       boost::unique_lock<boost::mutex> runLock(*runningMutex);
11       if(!interrupted)
12       {
13         heuristic->joinReadyList(this);
14         running = false;
15       }
16       boost::this_thread::restore_interruption ei(di);
17       while(!running) runningCV->wait(runLock);
18       switch(action)
19       {
20         case threadGenerate :
21           heuristic->generate(offst, intvl); break;
22         case threadEvaluate :
23           heuristic->evaluate(offst, intvl); break;
24         case threadSearch :
25           heuristic->singleThreadSearch(offst); break;
26         default :
27           assert(false); break;
28       }
29     }
30     catch(const boost::thread_interrupted&)
31     {
32       if(running)
33       {
34         boost::unique_lock<boost::mutex> runLock(*runningMutex);
35         heuristic->joinReadyList(this);
36         running = false;
37       }
38       boost::unique_lock<boost::mutex> intLock(*intrrptMutex);
39       interrupted = true;
40       intrrptCV->notify_one();
41       if(heuristic->isDestructing()) break;
42     }
43   }
44 }
```

Listing B.1: The loop for worker thread execution in `HeuristicThread`.

B.2.3 Timer Thread

The job of the timer thread is to notify the original thread (which controls the search) that the time limit for the search has been reached. To perform this job, the timer thread needs to be activated. When a new search is started, the heuristic search object calls `TimerThread`'s method `wakeUp()` which has an argument for the number of milliseconds allocated in the search budget. The timer thread wakes up from its blocked sleep and then uses a `sleep()` method provided by the BOOST thread library.

When the timer thread awakens from the BOOST thread library sleep, it invokes a sequence of three methods provided by `HeuristicSearchMTE`:

1. `setAction(heurFinished)`, which tells the control thread that the next action is to finish the heuristic search,

2. `setStopFlag()`, which sets a flag that will cause the control thread to break out of the main search loop, and

3. `notifyActionComplete()`, which wakes up the control thread and notifies it that an action needs to be taken.

The timer thread then returns to its blocked state waiting for notification that a new search is beginning.

B.2.4 Destructing Threads

The destructor method for `HeuristicSearchMTE` collects the pointers for all the semaphores and condition variables used by the threads and stores them in temporary lists. The destructor method then sets the flag for `destructing` to true. The destructor method then interrupts all the threads and calls a `join()` via the BOOST thread library.

Similar to worker threads, the timer thread contains an exception handling block for interruption. Like the worker thread, it checks the `destructing` flag. If the `destructing` flag is set to true, both timer and worker threads cease execution.

B.2.5 Evaluation of Solutions

When a solution is presented to the `HeuristicSearchMTE` `evaluate()` method, a thread-safe hash table, `previouslyEvaluated`, is consulted. If the `solution` is present in `previouslyEvaluated`, the score of the previous solution is read from the hash table and applied to the current copy of the previously seen `solution`. A counter called `numConsecutiveCacheHits` is incremented. There is a limit to the number of consecutive cache hits allowed to prevent the search from entering an infinite loop when an evaluation budget is used (i.e., no timer thread is present to end the search).

If a solution is not found in `previouslyEvaluated`, `numConsecutiveCacheHits` is set to 0. The `evaluate()` method then calls the `solution`'s `score()` method. The `score()` method for an `Architecture` object will invoke a service selection search. The `score()` method for a `ServiceSelection` object will conduct model evluations and utility calculations. Eventually both architecture and service selections return a U_g as the score to the `evaluate()` method. The `solution` is then stored in `previouslyEvaluated`. If the `solution`'s score is greater than the `bestOverallScore`, `bestOverallFound` is assigned to the current solution. The `bestOverallScore` is updated as well. The `evaluate()` method will also check to see if the `bestOverallScore` is the maximum score possible. If the answer is yes, the search will terminate. (In this dissertation, the only times the maximum value was reached were during the `online SVM meta-controller`'s search for optimal SVM parameters.)

B.3 Local Search Implementation

The logic for hill-climbing, beam search, and simulated annealing described in Section 2.3.4 are implemented in their respective heuristic search templates . Similar to the timer thread setting the `heurFinished` action, worker threads can inform the control thread that new actions are warranted. The possible enumerated actions are:

- `heurWait`, which indicates the search has begun but no action has been decided yet,
- `heurEval`, which indicates that the current action is to evaluate `solutions`,

- `heurMove`, which indicates that the current action is to visit a new `solution` and generate a new neighborhood,
- `heurRestart`, which indicates that the current search should be restarted at a new randomly generate `solution`, and
- `heurFinish` (discussed with the timer thread in Section B.2.3).

Neighborhood generation in hill-climbing and simulated annealing is single-threaded. Neighborhood generation is conducted by a method implemented by the `solution` class. One of the goals of the heuristic search template design is to contain the complexity of mult-threading within the heuristic search templates; this goal makes multi-threaded neighborhood generation difficult. The design decision to forgo multi-threading in neighborhood generation could be re-visited in future versions.

Neighborhood generation in beam search can utilize up to k (the beam width) threads, since separate `solution` objects will be generate the neighborhoods. It is important to make sure that data structures shared between the `solutions` would not be thread-safe.

During neighborhood generation, the local search algorithms make use of the `torch` abstract data type for both architecture and service selection search. The `torch` contains what would typically be temporary data structures. However, these data structures can be large, and re-allocating and re-building these data structures is expensive. These data structures are retained in the `torch` class and provided in the argument of `solution`'s `getNeighborhood()`. Beam search must make k copies of the `torch` object as the data structures contained in the `torch` are not thread-safe.

When the `getNeighborhood()` method finishes, the `neighborhood` (a `Set` of `solutions`) is returned to the heuristic search. The heuristic search can take full advantage of the worker threads for evaluating the `solutions` in the `neighborhood`.

In opportunistic hill-climbing and simulated annealing, a worker thread may find a new `solution` that warrants a visit. When this happens, the worker thread will set the `action` to `heurMove`, and then notify the control thread. The control thread will interrupt the rest of the worker threads. After verifying that each worker thread received and processed the interrupt, the control thread proceeds to neighborhood generation of the new `solution`.

B.4 Evolutionary Algorithms

The templates `EvolutionaryProgrammingMTE` and `GeneticAlgorithmMTE` both inherit from the template `EvolutionaryAlgorithmMTE` (see Fig. B.1). The abstract template `EvolutionaryAlgorithmMTE` uses multiple inheritance from the templates `EvolutionaryAlgorithm` and `HeuristicSearchMTE`. The `EvolutionaryAlgorithm` template implements life cycle methods that are used to support inheriting template classes. The `EvolutionaryAlgorithm` template uses wrapper classes, called `Indviduals` (distinct from the wrapper classes discussed in Section B.1), around the abstract data type `solution`. This wrapper simplifies keeping copies of `solution` in case mutation or crossover operations yield an invalid `solution`. Also, the `Individual` wrapper class provides some functionality not shown in Fig. B.1 that enable certain efficiencies when two or more `solutions` in the population are identical.

The template `EvolutionaryAlgorithmMTE` takes the life cycle support methods provided by `EvoluationaryAlgorithm` to create new methods that enable worker threads to generate new `solutions` and evaluate `solutions`.

The templates `EvolutionaryProgrammingMTE` and `GeneticAlgorithmMTE` then implement their search procedures according to the logic presented in Section 2.3.5 using the support functions provided by `EvolutionaryAlgorithmMTE` and `EvolutionaryAlgorithm`.

Bibliography

Bibliography

[1] G. Abbas, A.K. Nagar, H. Tawfik, and J. Y. Goulermas. Quality of service issues and nonconvex network utility for inelastic services in the internet. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '09)*, pages 537–547, London, United Kingdom, September 2009.

[2] A. Agrawal, G. Karsai, and F. Shi. Generative programming via graph transformations in the model-driven architecture. In *OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture*, pages 229–240, Seattle, WA, November 2002.

[3] A. Agrawal, G. Karsai, and F. Shi. Graph transformations on domain-specific models. Technical report, Institute for Software Integrated Systems, November 2003.

[4] J. Almeida, V. Almeida, D. Ardagna, C. Francalanci, and M. Trubian. Resource management in the autonomic service-oriented architecture. In *Proc. 3rd IEEE International Conference on Autonomic Computing (ICAC 06)*, pages 84–92, Dublin, Ireland, June 2006.

[5] T. Bäck, F. Hoffmeister, and H. Schwefel. A survey of evolution strategies. In *4th International Conference on Genetic Algorithms*, pages 2–9, San Diego, CA, July 1991.

[6] 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.

[7] M. N. Bennani and D. A. Menascé. Resource allocation for autonomic data centers using analytic performance models. In *Proc. 2nd IEEE International Conference on Autonomic Computing (ICAC '05)*, pages 229–240, Seattle, WA, June 2005.

[8] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2006.

[9] A. Bivens, C. Chhuor, D. Dillenberger, G. Ferris, J. Fenton, and W. Chou. Autonomic load balancing, part 1: Cisco content switching module. http://www.ibm.com/developerworks/library/ac-ewlmload1/index.html, April 2006.

[10] M. B. Blake. Decomposing composition: Service-oriented software engineers. *IEEE Software*, 24:68–77, November 2007.

[11] D. Breitgand, R. Cohen, A. Nahir, and D. Raz. On fully distributed adaptive load balancing. *Lecture Notes in Computer Science*, 4785:74–85, September 2007.

[12] R. Calinescu, Lars 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.

[13] R. D. Callaway, M. Devetsikiotis, Y. Viniotis, and A. Rodriguez. An autonomic service delivery platform for service-oriented network environments. *IEEE Transactions on Services Computing*, 3(2):104–115, April 2010.

[14] 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.

[15] V. Cardellini, E. Casalicchio, V. Grassi, F. L. Presti, and R. Mirandola. QoS-driven runtime adaptation of service oriented architectures. In *7th Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 131–140, Amsterdam, The Netherlands, August 2009.

[16] Emiliano Casalicchio, Daniel A Menascé, Vinod Dubey, and Luca Silvestri. Optimal service selection heuristics in service oriented architectures. In *Quality of Service in Heterogeneous Networks*, pages 785–798. Springer, 2009.

[17] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[18] Y. Cheng, A. Leon-Garcia, and I. Foster. Toward an autonomic service management framework: A holistic vision of soa, aon, and autonomic computing. *IEEE Communications Magazine*, 46(5):138–146, May 2008.

[19] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[20] Andrea 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, September 2006.

[21] E. Dashofy, A. van der Hoek, and R. N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proc. 24th International Conference on Software Engineering*, pages 266–276, Orlando, FL, May 2002.

[22] M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *Proc. of the 1st Workshop on Self-Healing Systems*, pages 21–26, Charleston, SC, November 2002.

[23] J. Davis. GME: The generic modeling environment. In *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 82–83, Anaheim, CA, October 2003.

[24] Kenneth DeJong. *Evolutionary Computation*. MIT, Cambridge, MA, 2002.

[25] Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung. Self-managing systems: A control theory foundation. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS '05)*, pages 441–448, Greenbelt, Maryland, April 2005.

[26] Y. Diao, C. W. Wu, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. Parekh, C. Garcia-Arellano, M. Carroll, L. Chu, and J. Colaco. Comparative studies of load balancing with control and optimization techniques. In *Proc. of the American Control Conference*, pages 1484–1490, Portland, OR, June 2005.

[27] S. Dobson, S. Denazis, A. Fernández, D. Gaiti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 1:223–259, December 2006.

[28] C. Domeniconi, J. Peng, and D. Gunopulos. Locally adapative metric nearest-neighbor classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(9):1281–1285, September 2002.

[29] M. Dorigo and T. Stützle. *Ant Colony Optimization*. Bradford Books, Cambridge, Massachusetts, 1990.

[30] 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.

[31] J. J. Dujmović. Continuous preference logic for system evaluation. *IEEE Transactions on Fuzzy Systems*, 15:1082–1099, December 2007.

[32] J. J. Dujmović. Characteristic forms of generalized conjunction/disjunction. In *2008 IEEE International Conference on Fuzzy Systems (FUZZ 2008)*, pages 1075–1080, Hong Kong, China, June 2008.

[33] Ahmed 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, New York, NY, USA, 2010. ACM.

[34] Ahmed Elkhodary, Naeem Esfahani, and Sam 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, New York, NY, USA, 2010. ACM.

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

[36] 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, October 2009.

[37] 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, September 2009.

[38] J. M. Ewing and D. A. Menascé. Autonomic metaheuristic optimization with application to run-time software adaptation. In *The Eleventh International Conference on Autonomic and Autonomous Systems (ICAS)*, page in publication, Rome, Italy, May 2015.

[39] John M Ewing and Daniel 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*, pages 173–184. ACM, 2014.

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

[41] M. Fuchs. Learning proof heuristics by adapting parameters. In *Proc. Twelfth International Conference in Machine Learning (ML-95)*, pages 235–243, Tahoe City, CA, July 1995.

[42] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, January 2003.

[43] 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, March 2010.

[44] H. Gomaa and M. Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *Proc. 4th Working IEEE/IFIP Working Conference on Software Architecture*, pages 79–88, Oslo, Norway, June 2004.

[45] Hassan Gomaa and Koji Hashimoto. Dynamic software adaptation for service-oriented product lines. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, page 35. ACM, 2011.

[46] Hassan Gomaa and Koji Hashimoto. Dynamic self-adaptation for distributed service-oriented transactions. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 11–20. IEEE Press, 2012.

[47] Hassan Gomaa and Koji Hashimoto. Model-based run-time software adaptation for distributed hierarchical service coordination. In *ADAPTIVE 2014, The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications*, pages 1–6, 2014.

[48] Igor Griva, Stephen G. Nash, and Ariela Sofer. *Linear and Nonlinear Optimization*. Society for Industrial Mathematics, Philadelphia, PA, second edition, 2008.

[49] Koji Hashimoto. *Software Adaptation Patterns for Service-Oriented Architectures*. PhD thesis, George Mason University, 2010.

[50] T. Heinis, C. Pautasso, and G. Alonso. Design and evaluation of an autonomic workflow engine. In *Proc. 2nd IEEE International Conference on Autonomic Computing (ICAC '05)*, pages 27–38, Seattle, WA, June 2005.

[51] Paul Hensgen. Umbrello uml modeller. https://umbrello.kde.org/. version 2.15.2.

[52] K. Herrmann, G. Mühl, and K. Geihs. Self-management: The solution to complexity or just another problem. *IEEE Distributed Systems Online*, 6(1):1, January 2005.

[53] T. Horvath, K. Skadron, and T. Abdelzaher. Enhancing energy efficiency in multi-tier web server clusters via prioritization. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–6, Long Beach, CA, March 2007.

[54] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification, 2003.

[55] Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multiclass support vector machines. *Neural Networks, IEEE Transactions on*, 13(2):415–425, 2002.

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

[57] IBM. An architectural blueprint for autonomic computing. http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf, June 2006. 4th edition.

[58] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[59] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, New York, 1991.

[60] 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.

[61] I. J. Jureta, S. Faulkner, Y. Achbany, and M. Saerens. Dynamic web service composition within a service-oriented architecture. In *IEEE International Conference on Web Services (ICWS '07)*, pages 304–311, Salt Lake City, UT, July 2007.

[62] James Kennedy and Russell C. Eberhart. *Swarm Intelligence*. Morgan Kaufmann, San Francisco, 2001.

[63] J. O. Kephart and W. E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Proc. 5th International Workshop on Policies for Distributed Systems and Networks*, pages 3–12, Yorktown Heights, New York, June 2004.

[64] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.

[65] 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, September 2003.

[66] S. Kounev, F. Brosig, N. Huber, and R. Reussner. Towards self-aware performance and resource management in modern service-oriented systems. In *2010 IEEE International Conference on Services Computing (SCC)*, pages 621–624, Miami, FL, July 2010.

[67] Anne Koziolek, Heiko Koziolek, and Ralf Reussner. Peropteryx: automated application of tactics in multi-objective software architecture optimization. In *QoSA-ISARCS '11*, pages 33–42, New York, NY, USA, 2011. ACM.

[68] J. Kramer and J. Magee. Analyzing dynamic change in software architectures: A case study. In *Proc. 4th IEEE International Conference on Configurable Distributed Systems*, pages 91–100, Annapolis, MD, May 2007.

[69] 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.

[70] G. H. Kuenning. Mersenne twist pseudorandom number generator package. http://lasr.cs.ucla.edu/geoff/mtwist.html, July 2008. version 1.5.

[71] D. Kusic and N. Kandasamy. Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems. *Cluster Computing*, 10(4):395–408, August 2007.

[72] J. W. Lee, R. R. Mazumdar, and N. B. Shroff. Non-convex optimization and rate control for multi-class services in the Internet. *IEEE/ACM Transactions on Networking*, 13(4):827–840, August 2005.

[73] W. S. Li, D. C. Zilio, V. S. Batra, M. Subramanian, C. Zuzarte, and I. Narang. Load balancing for multi-tiered database systems through autonomic placement of materialized views. In *Proc. IEEE International Conference on Data Engineering (ICDE06)*, page 102, Atlanta, GA, April 2006.

[74] Kwei-Jay Lin, Jing Zhang, Yanlong Zhai, and Bin Xu. The design and implementation of service process reconfiguration with end-to-end qos constraints in soa. *Service Oriented Computing and Applications*, 4(3):157–168, 2010.

[75] M. Lin, X. Jianshan, G. Heqing, and H. Wang. Solving QoS-driven web service dynamic composition as fuzzy constraint satisfaction. In *Proc. 2005 IEEE International Conference on e-Technology, e-Commerce, and e-Service (EEE'05)*, pages 9–14, Hong Kong, China, April 2005.

[76] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck. Web service level agreement (WSLA) language specification. http://researchweb.watson.ibm.com/wsla/WSLASpecV1-20030128.pdf, January 2001.

[77] S. Malek, N. Esfahani, D. A. Menascé, J. Sousa, and H. Gomaa. Self-architecting software systems (SASSY) from QoS-annotated models. In *Principles of Engineering Service Oriented Systems (PESOS '09)*, pages 62–69, Vancouver, Canada, May 2009.

[78] Sam Malek, Marija Mikic-Raki, and Nenad Medvidovic. A decentralized redeployment algorithm for improving the availability of distributed systems. *LNCS*, 3798:99–114, November 2005.

[79] Sam Malek, Marija Mikic-Raki, and Nenad Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31(3):256–272, March 2005.

[80] E. Mancini, M. Rak, and U. Villano. A simulation-based framework for autonomic web services. In *Proc. 11th International Conference on Parallel and Distributed Systems*, pages 433–437, Fukuoka, Japan, July 2005.

[81] Nariman Mani, Dorina C Petriu, and Murray 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.

[82] Anne Martens, Danilo Ardagna, Heiko Koziolek, Raffaela Mirandola, and Ralf Reussner. A hybrid approach for multi-attribute QoS optimisation in component based software systems. In GeorgeT. Heineman, Jan Kofron, and Frantisek Plasil, editors, *Research into Practice–Reality and Gaps*, volume 6093 of *LNCS*, pages 84–101. Springer Berlin Heidelberg, 2010.

[83] M. Marzolla and R. Mirandola. Performance aware reconfiguration of software systems. Technical report, The University of Bologna, May 2010.

[84] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transtions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[85] D. A. Menascé and V. Akula. A business-oriented load dispatching framework for online auction sites. In *Proc. 4th IEEE International Conference on Quantitative Evaluations of Systems (QEST07)*, pages 249–258, Edinburgh, Scotland, September 2007.

[86] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. *Performance by Design: Computer Capacity Planning by Example*. Prentice Hall, Upper Saddle River, NJ, 2004.

[87] 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.

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

[89] 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, October 2001.

[90] D. A. Menascé and V. Dubey. Utility-based QoS brokering in service oriented architectures. In *Proc. IEEE International Conference on Web Services (ICWS '07)*, pages 422–430, Salt Lake City, UT, July 2007.

[91] D. A. Menascé and V. Dubey. On composing and decomposing QoS goals in service oriented architectures. In J. Suzuki, editor, *Methodologies for Non-Functional Requirements in Serive Oriented Architectures*. IGI Global, 2010.

[92] 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, January 2010.

[93] D. A. Menascé, H. Ruan, and H. Gomaa. QoS management in service-oriented architectures. *Journal of Performance Evaluation*, 64(7–8):636–663, August 2007.

[94] 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.

[95] Daniel A Menascé, Emiliano Casalicchio, and Vinod Dubey. On optimal service selection in service oriented architectures. *Performance Evaluation*, 67(8):659–675, 2010.

[96] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer, New York, second edition, 2004.

[97] M. Mikic-Rakic, S. Malek, and N. Medvidovic. Improving availability in large, distributed component-based systems via redeployment. *LNCS*, 3798:83–98, November 2005.

[98] A. Missaoui and K. Barkaoui. A neuro-fuzzy model for QoS based selection of web service. *Journal of Software Engineering and Applications*, 4(3):588–592, June 2010.

[99] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In D. Scott, editor, *Artificial Intelligence: Methodology, Systems, and Applications*, volume 2443 of *LNCS*, pages 389–409. Springer Berlin, Heidelberg, 2002.

[100] O. Nano and A. Zisman. Realizing service-centric software systems. *IEEE Software*, 24(6):28–30, November 2007.

[101] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proc. 20th International Conference on Software Engineering*, pages 177–186, Kyoto, Japan, April 1998.

[102] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40(11):38–45, November 2007.

[103] C. Pautasso, T. Heinis, and G. Alonso. Autonomic execution of web service compositions. In *Proc. 2005 IEEE International Conference on Web Services (ICWS '05)*, page 435, Orlando, FL, July 2005.

[104] C. Pautasso, T. Heinis, and G. Alonso. JOpera: Autonomic service orchestration. *IEEE Data Engineering Bulletin*, 29(3):32–39, September 2006.

[105] D. Perez-Palacin and J. Merseguer. Performance evaluation of self-reconfigurable servie-oriented software with stochastic petri nets. *Electronic Notes in Theoretical Computer Science*, 261:181–201, February 2010.

[106] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated composition of web services by planning at the knowledge level. In *19th International Joint Conference on Artificial Intelligence*, pages 1252–1259, Edinburgh, Scotland, August 2005.

[107] 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.

[108] V. Poladian, D. Garlan, M. Shaw, M. Satyanarayanan, B. Schmerl, and J. Sousa. Leveraging resource prediction for anticipatory dynamic configuration. In *Proc. 1st IEEE Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 214–223, Boston, MA, July 2007.

[109] Bhavani Raskutti and Adam Kowalczyk. Extreme re-balancing for svms: a case study. *ACM Sigkdd Explorations Newsletter*, 6(1):60–69, 2004.

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

[111] James Robergé. *Data Structures in C++: A Laboratory Course*. Jones and Bartlett Publishers, Sudbury, MA, 1997.

[112] R. T. Rockafellar. Lagrange multipliers and optimality. *SIAM Review*, 35(2):183–238, June 1993.

[113] Jonathan Rowe, Darrell Whitley, Laura Barbulescu, and Jean-Paul Watson. Properties of gray and binary representations. *Evolutionary Computation*, 12(1):47–76, 2004.

[114] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, second edition, 2002.

[115] M. Schmid and R. Kroeger. Decentralised QoS-management in service oriented architectures. In R. Meier and S. Terzis, editors, *Distributed Applications and Interoperable Systems*, volume 5053 of *LNCS*, pages 44–57. Springer, Berlin, 2008.

[116] S. Shenker. Fundamental design issues for the future Internet. *IEEE Journal on Selected Areas in Communications*, 13(7):1176–1188, September 1995.

[117] S. F. Smith. Flexible learning of problem solving heuristics through adaptive search. In *Proc. Eighth International Joint Conference on Artificial Intelligence IJCAI'83*, pages 422–425, Karlsruhe, Germany, August 1983.

[118] J. P. Sousa, V. Poladian, D. Garlan, B. Schmerl, and M. Shaw. Task-based adaptation for ubiquitous computing. *IEEE Transactions on Systems, Man, and Cybernetics*, 36(3):328–340, May 2006.

[119] J. P. Sousa, Z. Zengin, and S. Malek. Towards multi-design of situated service-oriented systems. In *Proc. 2nd International Workshop on Principles of Engineering (PESOS '10)*, pages 57–63, Cape Town, South Africa, May 2010.

[120] R. Srikant. *Mathematics of Internet Congestion Control*. Birkäuser, Boston, MA, 2004.

[121] M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. *SIGPLAN Not.*, 38(5):77–90, 2003.

[122] G. Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, Wellesley, MA, 1993.

[123] Bjarne Stroustrup. Abstraction and the c++ machine model. In Zhaohui Wu, Chun Chen, Minyi Guo, and Jiajun Bu, editors, *Embedded Software and Systems*, volume 3605 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 2005.

[124] 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)*, pages 65–73, Dublin, Ireland, June 2006.

[125] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proc. International Conference on Autonomic Computing (ICAC '04)*, pages 70–77, New York, NY, May 2004.

[126] G. Wang, C. Wang, A. Chen, H. Wang, C. Fung, S. Uczekaj, Y. Chen, W. Guthmiller, and J. Lee. Service level management using QoS monitoring, diagnostics, and adaptation for networked enterprise systems. In *Ninth IEEE International EDOC Enterprise Computing Conference*, pages 239–248, Enschede, The Netherlands, September 2005.

[127] D. A. Waterman. Generalization learning techniques for automating the learning of heuristics. *Artificial Intelligence*, 1(1-2):121–170, April 1970.

[128] Anthony Williams and V. J. Botet Escriba. The boost thread library. http://www.boost.org/doc/libs/1_47_1/libs/boost_thread/boost_thread. htm), 2011.

[129] David H Wolpert and William G Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.

[130] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. On the use of fuzzy modeling in virtualized data center management. In *Proc. 4th IEEE International Conference on Autonomic Computing (ICAC '07)*, pages 25–34, Jacksonville, FL, June 2007.

[131] J. Zhang and R. J. Rigueiredo. Autonomic feature selection for application classification. In *Proc. 3rd IEEE International Conference on Autonomic Computing (ICAC '06)*, pages 43–52, Dublin, Ireland, June 2006.

[132] L. Zhang and D. Ardagna. SLA based profit optimization in autonomic computing systems. In *Proc. 2nd International Conference on Service Oriented Computing (ICSOC04)*, pages 173–182, New York, NY, November 2004.

[133] Y. Zhang, K. Lin, and J. Hsu. Accountability monitoring and reasoning in service-oriented architectures. *Service Oriented Computing Applications*, 1(1):35–50, June 2007.

Curriculum Vitae

From 2001 to 2005, John was employed as a computer scientist at the Defense Information Systems Agency.

From 1999 to 2001, John as a member of the senior technical staff and Computer, Networks, and Software Inc.

From 1997 to 1998, John served as the computer lab manager of the Rice Campus at the Illinois Institute of Technology.

In 2003, John earned a Master of Science in Computer Science from Illinois Institute of Technology.

In 1997, John received Bachelor of Science in Chemistry from University of Richmond.

In 1993, John graduated from York Community High School in Elmhurst, Illinois.