MANAGEMENT OF UNCERTAINTY IN SELF-ADAPTIVE SOFTWARE

by

Naeem Esfahani

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. Sam Malek, Dissertation Director
______	Dr. Daniel A. Menascé, Committee Member
______	Dr. Hassan Gomaa, Committee Member
______	Dr. Rajesh Ganesan, Committee Member
______	Dr. Sanjeev Setia, Department Chair
______	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering
Date: ______	Summer Semester 2014 George Mason University Fairfax, VA

Management of Uncertainty in Self-Adaptive Software

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

By

Naeem Esfahani
Master of Science
Sharif University of Technology, 2008
Bachelor of Science
University of Tehran, 2005

Director: Sam Malek, Associate Professor
Department of Computer Science

Summer Semester 2014
George Mason University
Fairfax, VA

Copyright © 2014 by Naeem Esfahani
All Rights Reserved

Dedication

هر جمعه دلم تازه ز سوگند قدیم　　«خواهم بشوم یار و هم راه فهیم»

ناچیز بود شأن نوشتار، زعیم　　تقدیم کنم بر قدمت جان نعیم

Acknowledgments

This dissertation would have not been possible without endless support from my lovely wife, Fahimeh Gilaki. I felt Fahimeh's encouragement, sacrifice, patience, and love, in every step of the way. Having Fahimeh on my side has been a blessing in my life.

I thank my parents, Mahvash Dana, Mina Tansaz, Abbas Esfahani, and Hassan Gilaki for their continuous support and prayers. I wish I could spend more time with them in recent years and I appreciate that they have put up with the long distance.

I would like to thank my adviser Dr. Sam Malek for his kind support and guidance. I never felt that Sam is sitting on the other side of the table; I have always seen him as my supportive friend. Sam's encouragements gave me the confidence to go the last mile.

I thank Dr. Daniel A. Menascé, Dr. Hassan Gomaa, and Dr. Rajesh Ganesan for serving in my committee and providing me with insightful feedback. I really appreciate their flexibility in scheduling my exams.

I thank Thabet Kacem, Ehsan Kouroshfar, and Kaveh Razavi for their helps. Finally, I thank my friends in software engineering lab for their good company.

Table of Contents

List of Tables

List of Figures

Abstract

MANAGEMENT OF UNCERTAINTY IN SELF-ADAPTIVE SOFTWARE

Naeem Esfahani, PhD

George Mason University, 2014

Dissertation Director: Dr. Sam Malek

The ever-growing complexity of software systems coupled with the need to maintain their quality of service (QoS) characteristics, even under adverse conditions and highly uncertain environments, have instigated the emergence of *self-adaptive* software systems [57]. A self-adaptive software system has the mechanisms that automate and simplify the management and modification of software systems after they are deployed, (i.e., during run-time) to achieve certain functional or QoS goals.

While the benefits of such systems are plenty, their development has shown to be significantly more challenging than static and predictable software systems [14, 59]. One key culprit is that self-adaptation is subject to *uncertainty* [14,59]. Uncertainty can be observed in every facet of adaptation, albeit at varying degrees. It follows from the fact that the system's user, adaptation logic, and business logic are loosely coupled, introducing numerous sources of uncertainty [16]. This challenges the confidence with which the adaptation decisions are made. A key observation is that while the level of uncertainty could vary, no self-adaptive software system is ever completely free of it.

This is precisely the challenge I addressed in this research. I have presented a general quantitative approach for tackling the complexity of automatically making adaptation decisions under uncertainty. I redefined the conventional definition of optimal solution to one that has the best range of behavior. In turn, the selected solution has the highest likelihood of satisfying the system's quality objectives, even if due to uncertainty, properties expected of the system are not borne out in practice.

In this dissertation, I begin with describing the problem and providing an overview of the solutions. Then, I define what I mean by uncertainty and position the approach with regard to the related work. Next, I provide the theoretical detail of the approach, which includes the formal specification of the problem and two (combinatorial and evolutionary) optimization techniques. I also describe some techniques for quantifying uncertainty and required step for effecting adaptation decisions at run-time. Finally, I discuss the implementation details of the framework. My experience with the approach, including experimental evaluation and porting the framework to a new problem domain, has been very positive. The evaluation results have strongly corroborated my hypotheses.

Chapter 1: Introduction

Self-adaptation is an effective approach in dealing with the changing dynamics of many application domains, such as mobile and pervasive systems. In response to changes in the environment or requirements, a self-adaptive software system modifies itself to satisfy certain objectives [14, 57, 59]. While the benefits of such systems are plenty, their development has shown to be more challenging than traditional software systems [14, 59]. One key culprit is that self-adaptation is subject to *uncertainty* [14, 59].

Currently, in the field of software engineering, uncertainty is considered as a second-order concept [41] and it is believed that by applying a set of practices the effect of uncertainty can be removed to allow focusing on the normal behavior. Although, it is generally true that having more information decreases the amount of uncertainty in a system [7], it is not possible to eliminate uncertainty altogether as it is not practical nor desirable to collect all the information about a system. In fact, no matter how much information is at hand, some things about the system may not be known. A key observation is that while the level of uncertainty could vary, it is rarely the case that a self-adaptive software system is completely free of uncertainty.

Uncertainty can be observed in every facet of adaptation, albeit at varying degrees. The key reason behind uncertainty is the fact that the system's user, adaptation logic, and business logic are loosely coupled, introducing numerous sources of uncertainty [16]. Consider that users often find it difficult to accurately express their quality preferences, sensors employed for monitoring often have uncontrollable noise, analytical models used for assessing the system's quality attributes by definition make simplifying assumptions that may not hold at run-time, and so on. These factors challenge the confidence with which the adaptation decisions are made. In spite of the fact that the uncertainty is prevalent in self-adaptive software systems, it is not studied in detail. Therefore, the term *'uncertainty'* has a very vague definition in the community; this affects the ability to work collaboratively to change the status quo.

The first contribution of this research is defining what I mean by uncertainty and describing what are the sources of uncertainty in self-adaptive software. This provided a common terminology for looking at the problems and communicating ideas. Moreover, I provided some background information that are required for understanding the solutions provided in this thesis. I also positioned my approach to the problem with regard to existing work in the community through providing the literature review.

The second and main contribution of this research is a general quantitative approach for tackling the complexity of automatically making adaptation decisions under uncertainty. I redefined the conventional definition of optimal solution to one that has the best range of behavior. This approach considered risks and opportunities associated with adaptation decisions to select the solution. In turn, the selected solution has the highest likelihood of satisfying the system's quality objectives, even if due to uncertainty, properties expected of the system are not borne out in practice.

These goals depend on collecting information at run-time for decisions-making. Therefore, I also provided a framework for quantifying uncertainty as a proof of concept. This paved the way for introducing the analysis technique that I used to make the decision in the face of uncertainty as part of the second contribution. I adopted a possibilistic analysis technique called *Possibilistic Linear Programming (PLP)* [50] to make such decisions. I also implemented an evolutionary version of the analysis, based on genetic algorithm, to deal with PLP's scalability problems.

I have applied this research to a dynamically reconfigurable robotic software system. The software components comprising this system are customizable, meaning that they can be configured to operate in different modes of operation. The configuration of a software component determines its quality attributes (e.g., response time) and resource usage (e.g., memory), which could also impact the properties of the entire system. However, as I mentioned earlier, these impacts are uncertain. The evaluation results demonstrate the ability to deal with uncertainty by making adaptation decisions that are superior to those of the conventional approach. I also used the same framework to deal with the problem of making early architectural decisions. Porting the framework to this new domain was a very successful experience.

The rest of this dissertation is organized as following. I first provide a motivating example in Chapter 2. In Chapter 3 I describe the problem and specify the scope of this thesis. Then I provide the overview of my approach in Chapter 4. In Chapter 5, I discuss the uncertainty in self-adaptive software. I provide the background and enumerates on the related work in Chapters 6 and 7 respectively. I formalize the problem in Chapter 8 and then describe how I quantified uncertainty in Chapter 9 as a prerequisite to solving the problem. I describe combinatorial optimization and evolutionary optimization as two possible techniques for finding the optimal solution in Chapters 10 and 11. In Chapter 12, I discuss how I realize an optimal solution in a running software. In Chapters 13 and 14, I describe the implementation of the framework and my experiments respectively. Finally, I conclude this dissertation with the description of a different problem domain that can benefit from the framework in Chapter 15 and the summary of the contributions in Chapter 16.

Chapter 2: Motivating Example

To demonstrate the ideas and help the discussion I used a robotic software system, that was developed in previous work [32], as a running example. The robotic software is part of a distributed search and rescue system [62] called *Emergency Deployment System (EDS).* EDS is aimed at supporting the government agencies in dealing with emergency crises (e.g., fire, hurricane). Figure 2.1b provides an abridged view of the robotic system's architecture. The software components comprising the robotic system range from abstractions of the physical entities, such as software controlled sensors and actuators on board the robot, to purely logical functionalities, such as image detection and navigation. The bold path in Figure 2.1b indicates the *Maneuver* execution scenario, which aims to safely steer the robot. The *Camera* feed is sent to *Obstacle Detector*, which runs an image processing algorithm to identify obstacles. Obstacle information is used by *Navigator* to plan the direction and speed of movement, which are then put into effect by the *Controller.*

The software components comprising this system are *customizable*, meaning that they can be configured to operate in different modes of operation. Figure 2.1a shows some of the available configuration dimensions. For instance, *Power* is a configuration dimension for the *Controller* component. A *Controller* could operate in either *Energy Saving* or *Full Power* mode. A component may have many configuration dimensions.

The configuration of a software component determines its quality attributes (e.g., response time) and resource usage (e.g., memory), which could also impact the properties of the entire system. For instance, given the resource-constrained nature of the mobile robots, the configuration decisions of each component have a significant impact on the system's performance as well as its battery life. Such decisions can only be effectively made at run-time, since the system properties (e.g., available bandwidth) are often not known at design-time and may change at run-time.

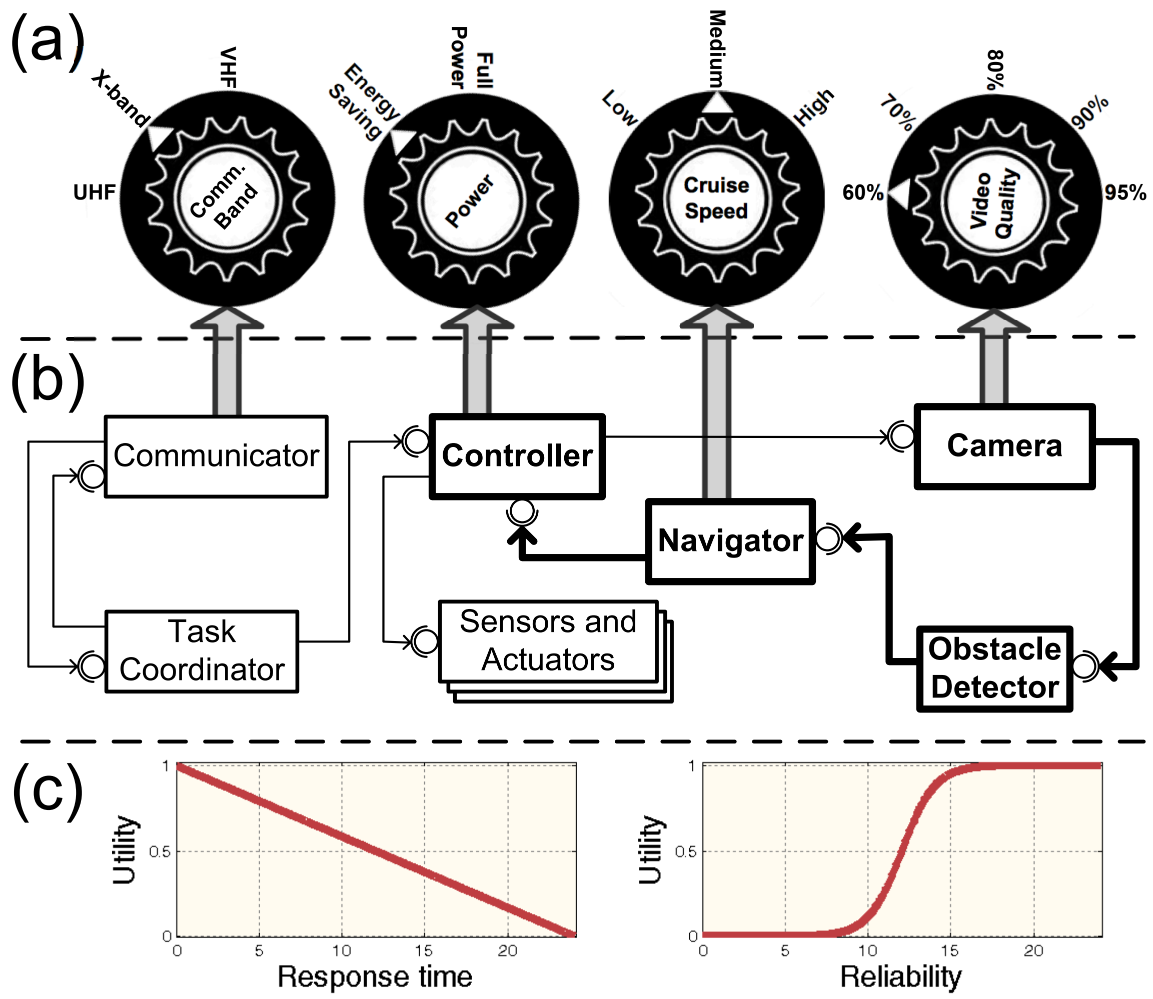


Figure 2.1: A subset of the robotic software: (a) configuration dimensions and alternatives for components of the robot, (b) software architecture, and (c) utility functions defined in terms of quality attributes.

As shown in Figure 2.1c, for making run-time decisions, utility functions capturing the user's satisfaction with different levels of quality attribute (e.g., availability) are used. The adaptation logic uses analytical models to estimate the effect of configuration decisions on the system's quality attributes, and in turn the resulting utility. For example, given the configuration of the robot's components, an analytical model, such as a Queueing Network model [68], may be used to quantify the response time of a particular scenario. The objective is to find a configuration that achieves the maximum overall utility.

The above approach is rather myopic, since it does not consider the uncertainty of information used in making adaptation decisions. Consider that almost every facet of the approach outlined above faces some form of uncertainty:

- **Uncertainty in System Parameters:** The monitoring data obtained from a running system rarely corresponds to a single value, but rather a distribution of values obtained over the observation period. For instance, a sensor monitoring the available network bandwidth may return a slightly different number every time a sample is collected. This variation could be either due to actual changes in the bandwidth or the error (noise) in the employed probes.

- **Uncertainty in Analytical Models:** Analytical models often make simplifying assumptions, and thus provide only estimates of the system's behavior. For instance, an analytical model quantifying the system's response time may account for the dominant factors, such as execution time of components, and ignore others, such as the transmission delay difference between TCP and UDP. Response time estimates provisioned by such a formulation are not only error-prone, but also the magnitude of error varies depending on the circumstances.

- **Uncertainty in User Preferences:** Eliciting user's preferences in terms of utility functions, such as those depicted in Figure 2.1c, is a well-known challenge [14]. Often users have difficulty expressing their preferences and thus the overall accuracy of the utility functions remains subjective, making the analysis based on them prone to uncertainty.

The uncertainty in these elements challenges the system's ability in making decisions that bring about the intended effects. Note that I picked these few sources of uncertainty among the many to describe the problem. I have provided a more detailed discussion of the sources of uncertainty in self-adaptive software in Section 5.2.

Chapter 3: Research Problem

In this chapter, I present the problem that I solved. In the following subsections, after describing the problem statement, I enumerate my research hypotheses.

3.1 Problem Statement

Self-adaptation endows a software system with the ability to satisfy certain objectives by automatically modifying its behavior. While many promising approaches for the construction of self-adaptive software systems have been developed, the majority of them ignore the uncertainty underlying the adaptation decisions. This has been one of the key inhibitors to widespread adoption of self-adaption techniques in risk-averse real-world settings, in which uncertainty is prevalent.

In my research, I focused on the improvement of software quality (i.e., non-functional requirements) through self-adaptation (recall Chapter 2). Therefore, I have scoped my study of uncertainty to how it affects the quality of software, or for that matter, adaptation decisions that are made based on the quality of software. In this scope, the uncertainty shows itself in inability to determine a single value for a property of interest. I refer to this as the *'uncertainty in self-adaptation parameters'*. I summarize the problem caused by uncertainty as following:

Due to uncertainty in a self-adaptive software system's parameters, the system may exhibit a range of behavior. Therefore, when uncertainty in the knowledge (models) used for making adaptation decisions is not accounted for in a self-adaptive software system, it may fail to satisfy the system's objectives. As the amount of the uncertainty in the knowledge used for decision-making increases, the chance of system failing to satisfy its objectives increases.

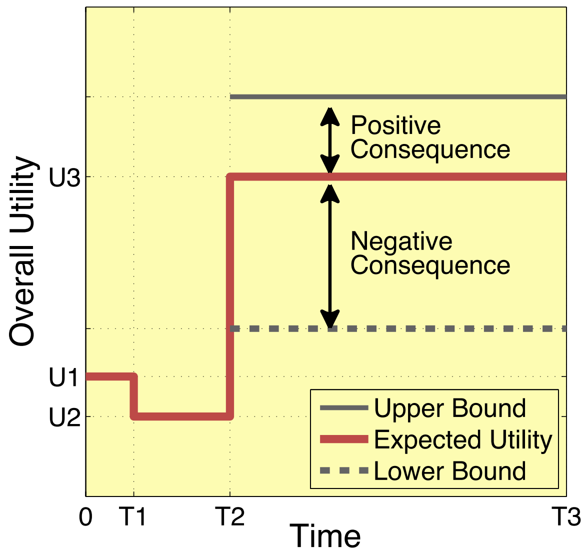


Figure 3.1: The utility of a self-adaptive system based on an adaptation decision depicted as a range.

Figure 4.1 depicts the problem with uncertainty. The system is initially executing with utility U_1 prior to time T_1. At time T_1, due to either an internal or external change, the system's utility drops to U_2. By time T_2, the self-adaptation logic detects this drop in utility, finds and effects an optimal configuration, which is conventionally defined as the one achieving the maximum utility. As shown in Figure 3.1, this corresponds to U_3, which represents the *expected* utility of the best configuration for the system.

However, the *actual* utility of the system may be different from the *expected* utility in practice. Figure 3.1 depicts the fact that the *actual* utility may vary between the *Upper* and *Lower* bounds, representing the likely positive and negative consequences of uncertainty (i.e., opportunity and risk) during T_3; by not accounting for the uncertainty in self-adaptation parameters in making decisions, the approach is vulnerable to gross overestimation of the utility. The following subsections discuss these ideas in more detail.

3.2 Research Hypotheses

This research has investigated the following hypotheses [30]:

- The uncertainty in the knowledge and information used to make adaptation decisions is inevitable. There are several mathematical approaches for dealing with uncertainty by considering a range of behavior instead of point estimates. These approaches have been successfully used in other fields (e.g., control theory, economics, and statistics).

  Hypothesis 1: *Considering ranges of uncertainty instead of point estimates, in decision making, results in adaptation decisions that transition the corresponding system into a state with higher quality after adaptation.*

- Uncertainty in making adaptation decisions is neglected partially due to the misconception that dealing with uncertainty is computationally expensive. However, there are modern mathematical approximation techniques that can deal with uncertainty efficiently.

  Hypothesis 2: *Uncertainty can be incorporated in the decision-making process in a manner suitable for execution at run-time.*

- Precisely quantifying uncertainty is a difficult task. Quantifying uncertainty "as it is" is a very hard task if not impossible.

  Hypothesis 3: *Incorporating partial information about the uncertainty in analysis allows for more accurate decisions compared to having no information about uncertainty at all (i.e., considering only point estimates).*

Chapter 4: Approach Overview

Figure 4.1a shows the typical behavior of a self-adaptive system that does not incorporate uncertainty in its analysis. I abstractly refer to this as the *traditional approach*. The centerpiece of my approach is the reconceptualization of what is traditionally considered as the optimal solution, such that the uncertainty is incorporated into the analysis. I illustrate the insights underlying this approach using Figure 4.1b. Similar to the scenario of Figure 4.1a, a new configuration is effected at time T_2, except my strategy is to select the configuration that concurrently satisfies the following three objectives: (1) maximizes U_3, which represents the most likely utility for the system under uncertainty; (2) maximizes the *positive consequence of uncertainty*, which represents the likelihood of the solution being better than U_3; and (3) minimizes the *negative consequence of uncertainty*, which represents the likelihood of the solution being worse than U_3.

I transformed these three concurrent objectives into a PLP problem, which is an instance of a multi-objective problem. To solve the PLP problem using commonly available linear programming solvers, I first transformed it to an equivalent single-objective problem. This was necessary to allow me to reason about the objective trade-offs. Intuitively the transformation process entails (1) normalizing the objective functions, (2) combining the objective functions, and (3) if necessary, specifying priorities among the objectives. This transformation was also necessary for solving the PLP problem using evolutionary optimization.

The details of the approach, including how the likelihood of each objective is calculated, are described in Chapters 9, 10, and 11. Nevertheless, I can make a general observation. As depicted in Figure 4.1, concurrent satisfaction of the three objectives may result in a smaller value of *expected* utility (i.e., U_3) in the approach compared to that of the traditional approach. But since the information used to estimate the *expected* utility is uncertain,

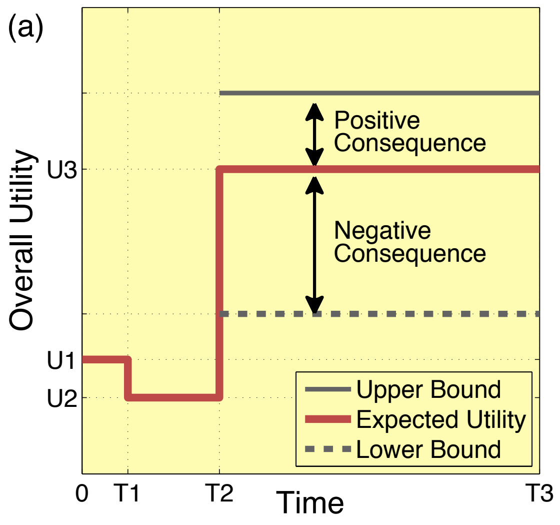


Figure 4.1: The utility of a self-adaptive system based on the decision using: (a) traditional approach, where the uncertainty is not considered and (b) my approach, which considers uncertainty.

expected utility is not guaranteed to occur in practice. I argue the true quality of a solution is determined by the range of possible utility. As depicted here and evaluated in Section 14, the objective has been to find solutions with a better range of behavior.

Based on this overall idea, I have developed a general quantitative approach for tackling the complexity of automatically making adaptation decisions under uncertainty [32]. In this framework, estimates of uncertainty in the elements comprising a self-adaptation problem are incorporated in possibilistic analysis of the adaptation choices. Therefore, I called this approach *POssIbilistic SElf-aDaptation (POISED)*. Unlike any other related work, POISED adopts a possibilistic method to assess the positive and negative consequences of uncertainty in its analysis. The centerpiece of my work is the reconceptualization of what is typically considered to be the optimal solution as one that has the best range of possible behavior (as depicted in Figure 4.1). In turn, the selected solution has the highest likelihood of satisfying the system's quality objectives, even if due to uncertainty, properties expected of the system are not borne out in practice.

Chapter 5: Uncertainty in Self-Adaptive Software

In this chapter I clarify as what I mean by uncertainty in self-adaptive software systems. I define the concept of uncertainty in self-adaptive software systems and then enumerate several sources of uncertainty in such systems. Finally, I provide a characterization of the presented sources of uncertainty.

5.1 Definition of Uncertainty

Making the adaptation decision is the centerpiece of a self-adaptive software system. Therefore, uncertainty in self-adaptive software means the lack of enough information or insight for making certain adaptation decisions. The adaptation decision can be made with certainty when all of the required information for making such decisions are available completely. However, usually some information are lacking during the decision making. Moreover, the available information may be according to approximate models, expert opinion, rules of thumb, and even pure intuition and hence inaccurate. In general it is hard to have the right amount of exact information at the right time. The sources of uncertainty indicated above more or less correspond to this definition.

All sources of uncertainty do not have the same characteristics. Although there are some philosophical debates about true distinction between different types of uncertainty (e.g., [90]), it is commonly agreed that it is useful to categorize different types of uncertainty in practice. This is due to the fact that the approaches for modeling different kinds of uncertainty are completely different. For instance, it is a mistake to model the uncertainty in the behavior of the system with a changing context over time as simple random process. In this scenario, resources, that are spent on collecting samples for estimating the random behavior, would be wasted as the pattern in the behavior of the system based on the execution context is ignored. Therefore, In the following subsections I enumerate different characteristics of uncertainty.

5.1.1 Reducibility versus Irreducibility

When something is inherently unknowable, the uncertainty associated with it is irreducible. On the other hand, the uncertainty associated with knowable things which are unknowns at the given time is reducible. Sometimes distinction between these two kinds of uncertainty becomes a philosophical problem, which depends on the point of view. One of the main reasons behind irreducible uncertainty is intractable complexity of phenomena with existing progress in science. For instance, it is a known fact that the physical world behaves in a non-linear fashion; however, there is little known about non-linear mathematics. Instead, non-linear phenomena are modeled using linear mathematics and hence the models have irreducible uncertainty. One may argue that this kind of uncertainty is not inherently irreducible as it can be mitigated by studying non-linear mathematics. I stay away from philosophical debates as I want to study the practical aspects of uncertainty.

5.1.2 Variability versus Lack of Knowledge

From a different perspective uncertainty can be categorized as aleatory or epistemic [7]. The root of aleatory is the Latin word ãleãtor, which means gambler, while the root of epistemic is the Greek word epistemé, which means scientific knowledge. Aleatory uncertainty captures the uncertainty that is caused by randomness and is usually modeled using probabilities. On the other hand, epistemic uncertainty corresponds to lack of knowledge and sometimes is referred to as parameter uncertainty. *This distinction is motivated by the location of the uncertainty — in the decision-maker or in the physical system.* [7] In other words, variability is considered as uncertainty in the studied system, while lack of knowledge is considered as uncertainty on the decision-maker's side.

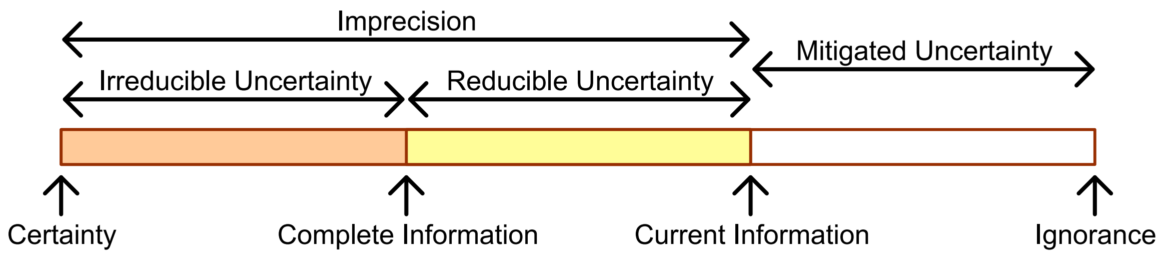


Figure 5.1: The horizon of uncertainty based on the knowledge, adapted from [7].

It may be tempting to map variability to irreducibility and lack of knowledge to reducibility. However, this is not generally true; for instance, if irreducible uncertainty directly implies variability, the next recipient of Turing Award, which in not known right now, would be a random phenomenon! Similar to the philosophical argument about reducibility versus irreducibility, there are arguments about distinction between aleatory and epistemic uncertainties. For instance, some argue that variability observed is the world is due to limitation of scientific models and hence lack of knowledge [90]. While these arguments are true, I want to mention that these distinctions are relative and depend on the point of view. In other words, it is true that sometimes a phenomenon, which is uncertain due to variability from a given point of view, can be uncertain due to lack of knowledge from a different point of view; but, this does not mean that variability is not a characteristics of uncertainty.

5.1.3 Adopted definition

Figure 5.1 depicts the horizon of uncertainty. *Current Information* falls anywhere between *Ignorance* and *Certainty*. The range between the *Current Information* and *Certainty* is the *Imprecision*. *Complete Information* indicates the threshold where all the knowable are known and falls anywhere between the *Current Information* and *Certainty* (i.e., inside *Imprecision*). In a sense, the *Complete Information* is a limit for the *Current Information* indicating the maximum amount that the uncertainty can be reduced. Therefore, the range between the *Current Information* and the *Complete Information* is the *Reducible*

Uncertainty. On the other hand, the range between the *Complete Information* and *Certainty* indicates the *Irreducible Uncertainty*.

Based on the nature of a given system, the length of any of these ranges (i.e., imprecision, reducible uncertainty, and irreducible uncertainty) can be zero. For instance, when the complete information and certainty point to the same spot, there will be no irreducible uncertainty. This definition also implies the fact that, as the current information increases and approaches the complete information, the imprecision becomes mainly due to irreducible uncertainty. Usually as the current information gets closer to the complete information, increasing the knowledge becomes more expensive. Sometimes increasing the knowledge may not even worth spending resources, as the added value becomes limited. I will come back to this issue in the next section.

Both the reducible and irreducible uncertainties can have aleatory and epistemic components. In other words, aleatory and epistemic represent the essence of uncertainty, while irreducible and reducible represent the managerial aspect of uncertainty. Managerial aspect of uncertainty does not give more insight into the its nature. Therefore, in the following subsection I only use the other aspect (i.e., the essence of uncertainty) to categorize the sources of uncertainty.

5.2 Sources of Uncertainty

Figure 5.2 depicts the high level view of a self-adaptive software system according to FORMS reference model [88]. In this model, the self-adaptive software system is broken down into two parts: *Meta-Level* and *Base-Level*. The base-level subsystem provides the main functionality of the software, while the meta-level subsystem manages the base-level subsystem by reflecting on it. Inside the meta-level subsystem there is a feedback control loop according to MAPE-K reference architecture [52] from IBM.

The other two entities in Figure 5.2 are *User* and *Environment*. The user uses the services of base-level subsystem and provides her expectations from the base-level subsystem to the meta-level subsystem by specifying objectives. For instance, Figure 2.1c shows users'

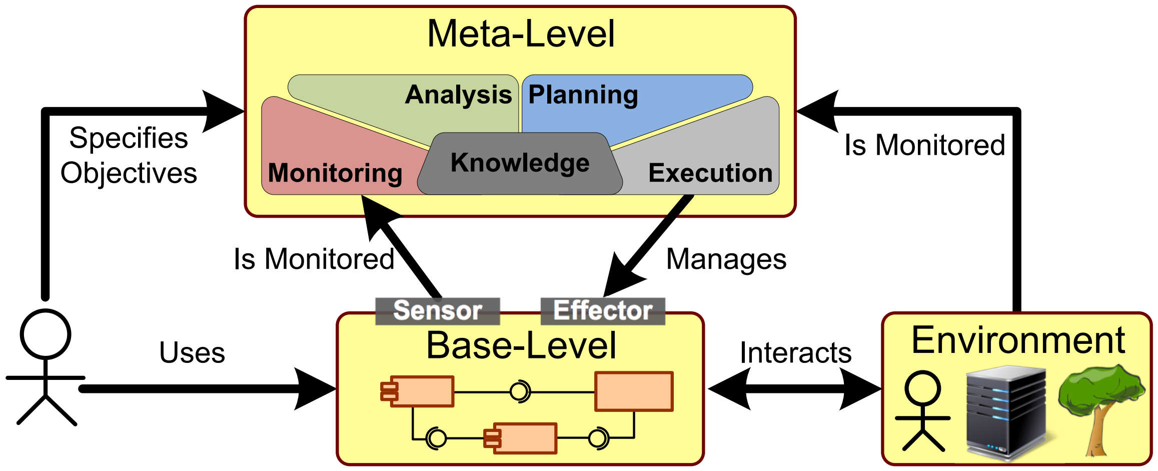


Figure 5.2: High level view of self-adaptive software.

expectation for the robotic software system in terms of two QoS parameters (i.e., Response Time and Reliability) of the *Maneuver* execution scenario; these expectations are depicted using utility functions. The self-adaptive software system operates in an environment and hence base-level subsystem constantly interacts with elements of that environment. Since the meta-level subsystem is responsible for keeping the base-level subsystem on track, it also needs to monitor the environment. For instance, in the robotic software system depicted in Figure 2.1, the meta-level subsystem uses sensors to estimate the amount of light in the operational environment to adjust the *Camera* accordingly.

The entities in Figure 5.2 are loosely coupled. The meta-level subsystem needs to use models of other entities in Figure 5.2 as their abstractions to make adaptation decisions. Therefore, one of the main places that the uncertainty lives is the interfaces of the meta-level subsystem with the other components. I discuss these sources of uncertainty as well as others in the following:

- **Uncertainty due to simplifying assumptions:** This source of uncertainty is related to the *"Manages"* interface in Figure 5.2 and is due to inaccuracy in the analytical models representing complex base-level subsystem. These analytical models are used to reason about the impact of adaptation choices on system's quality attributes.

The error in those estimates is magnified when the modeling abstractions become inaccurate representation of the system. One of the reasons for inaccuracy is the fact that sometimes the assumptions underlying the model are not held at run-time. For instance, an analytical model quantifying the system's response time may account for the dominant factors, such as execution time of components, and ignore others, such as the transmission delay difference between TCP and UDP. Response time estimates provisioned by such a formulation are not only error-prone, but also the magnitude of error varies depending on the circumstances.

- **Uncertainty due to drift:** This source of uncertainty is also related to the *"Manages"* interface. Another reason for inaccuracy in the analytical models is adaptation itself; the base-level subsystem may not enact the changes exactly as meta-level subsystem requested causing a drift from the models. For instance, in the same example, consider the scenario that the meta-level subsystem requests the base-level subsystem to change the protocol from TCP to UDP. However, the base-level subsystem fails to enforce this change, leaving the analytical models in the meta-level subsystem inconsistent with the behavior of the base-level subsystem.

- **Uncertainty due to noise:** This source of uncertainty corresponds to *"Is Monitored"* interfaces and is due to variation in a phenomenon, such as a monitored system parameter, which rarely corresponds to a single value, but rather a set of values obtained over the observation period. Consider that a sensor monitoring the available network bandwidth may return a slightly different number every time a sample is collected, while the actual value of the bandwidth may be fixed in the source. This type of uncertainty is referred to as noise to indicate the error in the employed probes.

- **Uncertainty of parameters over time:** This source of uncertainty is also related to *"Is Monitored"* interfaces and is due to the actual changes in the monitored phenomenon (e.g., bandwidth). In contrast to the previous source of uncertainty, consider the case that the actual bandwidth changes over time. Moreover, if the effects of adaptation alternatives in the future are not considered in decision making, the adaptation decision may behave unexpectedly in the future. For instance, I mentioned that the robotic software system may use sensors to measure the amount of light, which may change as the robot moves in the environment. This change can be predictable based on the trajectory of robot movement. If the robotic software system does not consider the prediction and make decision only based on the current amount of light, the adjustments to the *Camera* will not result in the expected improvement as the robot is moving.

- **Uncertainty due to human in the loop:** Self-adaptive software systems are increasingly permeating a variety of domains, including medical, industrial automation, and emergency response. This is partially caused by a paradigm shift from kinds of software deployed on isolated servers to the kinds of software that engages human users in their daily activities. These new breeds of software systems intensively interact with human and depend on correct human behavior. However, human behavior is inherently uncertain [41,85], which in turn creates uncertainty in the software system. This type of uncertainty is related to "Uses" interface between the base-level subsystem and the user. For instance, in the case of the robotic software system depicted in Figure 2.1, it is expected for the robot to interact with the rescue crew to fulfill its assignment. However, as described before the behavior of the crew may be very unpredictable.

- **Uncertainty in the objectives:** This type of uncertainty corresponds to the *"Specifies Objectives"* interface and is due to the complexity of users' requirements and preferences. In a large-scale multi-user system, users often have multiple concerns, some of which may be conflicting with one another. However, accurately eliciting and representing the users' preferences is a challenge. Thus, the overall accuracy of such preferences remains subjective, making the analysis based on them prone to uncertainty. For instance, eliciting user's preferences in terms of utility functions, such as those depicted in Figure 2.1c, is a well-known challenge [14] as the users often have difficulty expressing their preferences using mathematical functions.

- **Uncertainty due to decentralization:** In a self-organizing system several meta-level subsystems manage different base-level subsystems [59]. They build up a decentralized software system, where the knowledge is scattered among all the entities comprising the system. These entities can only access the interfaces of other entities and can hardly control the actions of other entities. In such a setting, the meta-level subsystems are expected to work collectively and collaboratively to reach the system's objectives. In other words, in self-organizing software systems, the meta-level subsystem is distributed among different entities, which makes the system prone to uncertainty. For instance, in the case of robotic software system, different robots may collaborate with each other to devise and update a plan for searching an area (e.g., a building that is damaged due to an earthquake) for victims with the goal of covering all the area as fast as possible. In other words, although the robots are autonomous and use self-adaptation to achieve their own functionality, they may need to collaborate with other robots to achieve the same goal. This high-level collaboration adds to uncertainty as the knowledge is decentralized.

- **Uncertainty in the context:** Many self-adaptive software systems are intended to be used in different contexts. To that end, it is expected from the meta-level subsystem to detect the change in the context and adapt the base-level subsystem to behave appropriately. Portable and embedded computing devices (e.g., cell-phones) are representative oof this class of systems. Here, software developers are forced to cope with additional sources of complexity introduced by the growing class of mobile and pervasive software, which are innately dynamic and unpredictable. The performance of these software systems heavily depends on availability of the resources [41], which is subject to change as the context of execution changes. For instance, in the robotic software system, a robot may move into a place in which a barrier shields its signal and prevents it to communicate with other robots; this makes the status of that robot unknown to the rest of system.

- **Uncertainty in cyber-physical systems:** As the computation becomes cheap and widespread, software and physical components become increasingly intertwined and cooperate more tightly. Therefore, it is expected from the meta-level subsystem to be able to manage physical entities as well. As a result, physical concepts, such as space and time, should also be considered in decision making. This increases non-determinism and uncertainty in the software due to the fact that the physical world itself is inherently uncertain. For instance, in the case of the robotic software system, assessment of distances from the obstacles is required for steering the robot safely. However, such assessments are physical in nature and can be uncertain.

To mitigate uncertainty in self-adaptive software systems one should consider its sources enumerated above. Some of these sources (e.g., cyber-physical systems) have been observed in other fields of science and there are well-established approaches for addressing them. On the other hand, some of these sources (e.g., drift) are relatively new and specific to self-adaptive systems, hence new approaches may need to be devised for addressing them.

I define the scope of this thesis with regard to the sources of uncertainty described here. Some of these sources are completely covered in this thesis and some are left for the future work. This thesis mainly address the uncertainty caused by *Simplifying assumptions*, *Noise*, and *Objectives*. I do not directly deal with the uncertainty related to *Drift*, *Human in the loop*, *Decentralization*, *Context*, and *Cyber-physical systems*. However, POISED can use the information provided about those sources of uncertainty to make adaptation decisions. In other words, although I do not provide any mechanism for quantifying uncertainty of those sources, POISED is general enough that it can use such provided information for making adaptation decision.

Table 5.1: Characteristics of different sources of uncertainty.

	Simplifying assumptions	Drift	Noise	Parameters over time	Human in the loop	Objectives	Decentralization	Context	Cyber-physical systems
Variability			✓	✓				✓	
Lack of Knowledge	✓	✓			✓	✓	✓		✓

5.3 Characteristics of the Sources of Uncertainty

Table 5.1 characterizes the sources of uncertainty based on their nature. To that end, I specify if a source of uncertainty is due to variability or lack of knowledge.

Uncertainty related to *Simplifying assumptions*, *Drift*, *Human in the loop*, *Objectives*, *Decentralization*, and *Cyber-physical systems* happens due to the lack of knowledge. Be it for the complexity of the models, loose coupling, ambiguity, or distribution of knowledge, the lack of complete knowledge makes the adaptation decisions prone to uncertainty.

On the other hand, uncertainty related to *Noise*, *Parameters over time*, and *Context* is due to the variability. In this case, uncertainty is rooted in the fact that the behavior of the system may change after the adaptation decision is made.

Chapter 6: Background

In this chapter I provide an overview of some fundamental concepts that knowing them is required for understanding this work.

6.1 Representing Uncertainty

The first step towards managing uncertainty is to represent it explicitly. Before representing uncertainty two important questions should be answered: (1) What is the right representation? and (2) How much information is required to build the representation?. These questions can be answered by considering the benefits and costs. The representation that has more information has less uncertainty. However, collecting information and building the representation is often an expensive task. Therefore, building complex representation only makes sense if the benefits exceed the costs of building such representation. To that end, similar to any other concept in software engineering, the appropriate level of abstraction should be considered in building the models of uncertainty. A good model contains enough information for making decisions without containing unnecessary information. In other words, the goal here is to build useful models of uncertainty for making informed decisions.

I start by exploring the second question first. Figure 6.1 depicts two hypothetical settings; in each setting an adaptation decision based on the quality metrics (e.g., reliability) of the base-level subsystem should be made. There may be several (in this case two) alternatives for an adaptation decision. The outcomes of an alternative (i.e., its effect on the quality metric) is uncertain, hence the quality under each alternative is a range rather than an exact number. Figure 6.1 depicts the ranges of the quality metric for different alternatives in the two hypothetical settings. Figure 6.1a shows the setting in which the

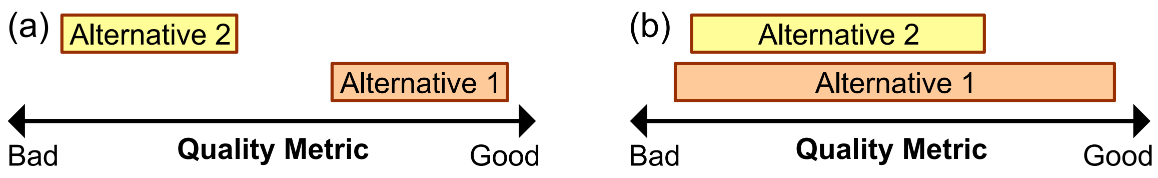


Figure 6.1: Comparing the quality metrics of an adaptation decision in the base-level subsystem for two alternatives in two hypothetical settings: (a) the model is precise enough and (b) the model needs to be refined with more information.

models are accurate enough for making the decision. As we can see the range of quality for *Alternative 1* is better than the range of quality for *Alternative 2*; this makes *Alternative 1* a better (e.g., more reliable) alternative for the decision. On the other hand, in the setting, which is depicted in Figure 6.1b, due to large amount of uncertainty, it is hard to distinguish between the two alternatives. Therefore, a meaningful decision requires more information to reduce the uncertainty.

Having said that, it may be the case that the quality of both alternatives in Figure 6.1b are above an acceptable threshold (e.g., both of them are reliable enough); this makes the selection between the two alternatives a less critical decision. This, in turn, may not justify spending resources for acquiring more information. Moreover, it may be the case that the uncertainty is irreducible and even acquiring more information would not help in making the trade-off between the two alternatives in Figure 6.1b.

In the following subsections I focus on the first question (i.e., what is the right representation?). To that end, I first describe how I quantify uncertainty in POISED and then I provide an overview of widely used mathematical approaches for modeling uncertainty in the literature.

6.1.1 Probability Theory

Probability theory [11] is the most widely used approach to represent uncertainty. Humans have long observed that some events are to some extent predictable. Mathematical probabilities, which are dated from 18th century, were an approach to study the regularities in the games of chance. Nowadays, probability is learned mainly through Kolmogorov's axioms [55], which allows for adoption of probability theory in broader class of problems (e.g., physical, social, industrial, etc.). Most researchers are familiar with the mathematics of probability but quiet few are aware of philosophical debates regarding different interpretations of probability. Therefore, here I focus on interpretations of probability. The prominent interpretations of probability until late twentieth century were classical and frequentist interpretations.

Probability theory was originally conceived with the classical interpretation. As I mentioned, probability was originally rooted in the games of chance, and so was the classical interpretation. A fundamental assumption in classical probability is the fact that all the outcomes of a phenomenon are equally probable. This assumption is shown to cause inconsistencies when it is used in more general problems (i.e., beyond games of chance). Motivated by the limitations of the classical interpretation, the frequentist interpretation was developed. In this interpretation the probability of an event is defined as limit of its relative frequency in large number of trials, hence the name of this interpretation is frequentist. Although this definition goes beyond classical definition, it narrows the scope of the frequentist interpretation to repeatable, random phenomena.

Bayesian theory [48] is based on subjective interpretation of the probability. In this interpretation the probability is defined as an expression of a rational agent's degrees of belief about uncertain propositions. The scope of this interpretation is more general than frequentist interpretation as it extends the definition of probability by allowing probability assignment to a single experiment regardless of whether it is part of a larger number of experiments or not. Therefore, Bayesian could be used in the problems in which there is not enough data for frequentist interpretation. For instance, frequentists cannot analyze a new disease for which enough data is not available, while Bayesians can use subjective information based on related diseases to analyze the new disease.

Bayesian inference is as old as probability. However, it was disfavored due to positive orientation of Western nineteenth and twentieth century science, which was considering subjectivity to be non-scientific. Moreover, complex Bayesian models require large amount of computation, which were not possible until late twentieth century. With computational advances in the late twentieth century there has been a resurgence towards Bayesian approaches as they are a unified theory for both data-rich and data-poor problems. Many modern machine learning methods are based on Bayesian principles.

6.1.2 Fuzzy Sets and Possibility Theory

Fuzzy set theory [91] is an extension of classical set theory. In classical set theory, the membership of an element in a set is a binary condition: the element is either in the set with membership value of 1 or it is not in the set with the membership value of 0. However, in fuzzy set theory, the membership of an element in a set is not a binary condition, but rather a “sort of” concept. To that end, the membership value of an element with regard to a set is any value between 0 and 1. The higher the membership value is, the more likely that element belongs to the set. Therefore, the boundary of a fuzzy set is not clearly defined, whereas the boundary of a classical set is *crisply* defined.

Fuzzy sets can be applied to domains where the information is incomplete or imprecise. For instance, fuzzy sets have been used in linguistics to deal with vagueness and ambiguity of the statements. For instance, temperatures that are considered to be cold and warm are not uniquely defined and they may be different from person to person. In fact, there are some temperatures that can be considered both cold and warm to some extent. A program that tries to understand written text can use the fuzzy definition of coldness and warmness to have a better understanding of the text.

Possibility theory [92] is a theory for handling incomplete information, which is based on fuzzy sets. Among several interpretations of possibility theory, the basic interpretation is the most common one. This interpretation defines possibility as a mapping from the power set of sample space to any value between 0 and 1. In other word, any event, which is a subset of sample space, has a possibility defined by this mapping. One of the reasons that fuzzy logic is adopted in engineering is the simplicity and efficiency of its operations.

While probability theory deals with the statistical characteristic of data, possibility theory focuses on the meaning of data. There are several studies [18, 24–26] about the relationships of the two theories. Although, sometimes the two theories can be used interchangeably, it has been shown that the two theories are different. Some researchers have described the usability of two theories using an spectrum: possibility theory is useful when there is little information, however, when more information becomes available it is better to use probability theory.

6.2 Analysis under Uncertainty

The discussion in Chapter 4 assumes one could quantify the range of utility under uncertainty. As I described in Section 6.1, there are two general techniques to estimating uncertainty: *probability theory* and *possibility theory*. Probability theory is concerned with the analysis of random phenomena and forms the foundation of statistics. Possibility theory is founded on the concept of fuzzy set [92]. In a fuzzy set, the elements have a degree of membership. Degree of membership is a value between zero and one: a value of zero indicates the element is certainly not a member of the set, a value of one indicates the element is certainly a member of the set, and a value in between indicates the extent of certainty that the element is a member of the set. In possibility theory, the concept of *possibility* is defined as the *degree of membership*, which plays a similar role as that of probability in statistics. The *most optimistic* and *most pessimistic* values have a degree of membership of zero, while the *most possible* value has a degree of membership of one. While the notion of probability and possibility are related, it is important to note that the two concepts are distinct. *Probability theory* deals with the statistical characteristic of data, while *possibility theory* focuses on the meaning of data [92].

For an illustration of this difference, see Figures 9.1 and 9.2 in which the uncertainty in available network bandwidth is modeled using possibility and probability distributions, respectively. The details of how such functions can be obtained in the first place are described in Section 9. Possibility distribution models a *fuzzy variable*, while probability distribution models a random variable. Possibility distribution in Figure 9.1 returns the *degree of membership*, while probability distribution in Figure 9.2 returns the *probability density*.

While uncertainty can be represented using both approaches, the ability to make adaptation decisions is significantly impacted by the representation. The operations research technique for making decisions under probability theory is called *stochastic programming* [2]. When all the probability distributions follow normal distribution, by applying the central limit theorem [44], the problem can be solved effectively [50]. Central limit theorem allows for the definition of commonly used algebraic operations on the normally distributed variables. However, as further demonstrated in this paper, uncertainty in self-adaptive software systems rarely follows a normal distribution, which severely limits the application of stochastic programming.

When some of the probability distributions are not normal, the problem is broken down into two parts and solved iteratively in two stages: a certain part, solved in the first stage, and an uncertain part, solved in the second stage. The two stages are executed iteratively similar to two nested loops. Given an assignment for the decision variables corresponding to the certain part, the remaining decisions variables corresponding to the uncertain part are solved, and the expected value of the objective function is calculated. In other words, for any given candidate solution to the first stage, an instance of the second stage optimization is configured and solved [4]. This approach is challenged by the following: (1) handling uncertain parameters in constraints of a stochastic programming problem is a matter of research (e.g., see [10]), and there is no generally applicable approach; (2) for each candidate solution to the first stage, an instance of second stage problem, often an NP-hard problem, needs to be solved; (3) calculating the expected value, which is required in the second stage is computationally expensive (e.g., requires integration over random distributions), and in some cases not even feasible, as shown in [4].

An alternative approach that does not suffer from the same shortcomings is *possibilistic programming*, which is founded on possibility theory. Possibilistic programming is widely used for making decisions under uncertainty in many fields of engineering, including control theory, robotics, and artificial intelligence. Advantages of possibilistic programming are twofold: generality and efficiency. Unlike stochastic programming, where simple algebraic operations require special considerations (e.g., central limit theorem), and not even always possible, in possibilistic programming, fuzzy variables can be simply operated on using traditional algebraic operators. This is true even if the distributions are complex and unique. Possibilistic programming problems can be solved much more efficiently than stochastic programming problems [50], making them desirable in many practical engineering problems, including self-adaptive systems. For these reasons I have adopted possibilistic programming as the technique for achieving the objectives.

While the characteristics of the problem made possibility theory a better fit, probability theory has shown to be suitable in many areas of software engineering. Thus, the conclusions made here are only with respect to the objectives. A more comprehensive analysis of the differences between the two theories can be found in the extensive body of literature [18, 24–26].

Chapter 7: Related Work

The research community has made great strides in tackling the complexity of constructing self-adaptive software systems [14,57,59]. However, as corroborated by others [14,59], there is a dearth of applicable techniques for handling uncertainty in this setting. The literature in this area of research is extensive. I refer the reader to existing surveys for a comprehensive analysis of the state-of-the-art in self-adaptation [77] and autonomic computing [49]. Here I describe the seminal and closely related work.

7.1 Foundations of Self-Adaptive Software

IBM's Autonomic Computing initiative advocates a reference model known as MAPE-K [52], which is structured as hierarchical levels of feedback-control loops. In each loop, there are four types of activities that operate on the managed subsystem and are devoted to *Monitoring*, *Analysis*, *Planning*, and *Execution* (*MAPE*). MAPE activities share various models using what is known as *Knowledge* (*MAPE-K*). At run-time, the activities, which build the feedback-loop, are performed in the following logical flow:

- **Monitor:** Collects data through instrumentation of the running system. If a functional failure or a violation of QoS objective is detected, it correlates the data into symptoms that can be analyzed.
- **Analyze:** When a problem is detected, it searches for a configuration that resolves it. It may perform a trade-off analysis between multiple conflicting goals.
- **Plan:** Chooses a path of adaptation steps towards the target configuration. The path has to abide by the system constraints. In addition, adaptation steps must not cause further failures in the system.

- **Execute:** Takes the required actions to effect the changes delineated in the plan. This may require adding, removing, and replacing the components and the way they are interconnected in the running architecture.

Previous studies have shown that a promising approach to resolve the challenges of constructing complex software systems is to employ the principles of software architecture [74, 79]. Software architectures provide abstractions for representing the structure, behavior, and key properties of a software system. They are described in terms of software components (computational elements), connectors (interaction elements), and their configurations. A given software architectural style (e.g., publish-subscribe, peer-to-peer, pipe-and-filter, client-server) further refines a vocabulary of component and connector types and a set of constraints on how instances of those types may be combined in a system [39].

Software architecture has also been shown to provide an appropriate level of abstraction and generality to deal with the complexity of dynamically adapting of software systems [57]. This observation has led to research on architecture-based adaptation, which is the process of reasoning about and adapting a system's software at the architectural level [57, 72]. Oreizy et al. pioneered the architecture-based approach to run-time adaptation and evolution management in their seminal work [72]. Tajalli et al. proposes an ADL based approach for dynamic planning and re-planning in response to unexpected circumstances using AI techniques [80]. Garlan et al. present Rainbow framework [42], a style-based approach for developing reusable self-adaptive systems. Rainbow monitors a running system for violation of the invariant imposed by the architectural model, and applies the appropriate adaptation strategy to resolve such violations. Kramer and Magee [57] define a three-layer model (component control, change management, and goal management) to address the challenges associated with the development of self-managed systems.

7.1.1 Monitoring and Effecting Adaptation

A generally applicable solution to the problem of functional consistency during adaptation was proposed by Kramer and Magee's seminal model of dynamic change management [56], which provides a separation of structural concerns from application concerns. A transaction is defined as an exchange of information (e.g., message, event) between two components, initiated by one of the components. Their work identifies two possible states for a software component during the adaptation process. Each state defines how a component behaves during the corresponding phase of adaptation. The two states are, (1) active: A component can start, receive, and process transactions. (2) passive: A component in this state will continue to receive and process transactions, but will not initiate any new transactions. Quiescence is defined as the required property of adapted component for a system to remain in a consistent state [56]. Quiescence implies that all the nodes that can initiate transactions on the updated component must be passive. This is known as the passive set [56].

Vandewoude et al. [86] propose Tranquility as a necessary condition for consistent adaptation of software systems. Tranquility builds on the notion of Quiescence proposed by Kramer and Magee, except it relaxes some of the constraints to achieve faster adaptation times. However, unlike Quiescence, Tranquility is not guaranteed to be reachable. Vandewoude et al. extended the Draco middleware [86] to provide support for Tranquility. However, Draco relies on software components to provide the middleware with not only a list of transactions they have already participated in the past, but also transactions they will participate in the future. This assumption breaks the black-box treatment of components, and is not practical, since it is not feasible for a third-party component to know a priori in which transactions it will participate.

Based on the existing models of dynamic change management [56], an approach [33] is introduced, which uses the rules and constraints of an architectural style to infer the component dependencies for any software system built according to that style. The component dependencies are in turn used to determine a reusable sequence of changes that need to occur for placing a component in the appropriate adaptation state. Such a recurring sequence of changes, which are coordinated among the system's architectural constructs (e.g., components, connectors) is called an adaptation pattern. An adaptation pattern provides a template for making changes to a software system built according to a given style without jeopardizing its consistency. Moreover, extracting this information from the architectural style prevents from violation of black box assumption regarding software components, which is often violated in the relate work (e.g., [56, 86]). In a subsequent work [35], these set of adaptation patterns are realized and extended on top of an existing middleware platform, called Prism-MW [65]. Gomaa et al. [46] applied a similar idea to the development of reconfiguration patterns for software product lines and they later extended their work to service oriented systems [45].

Prism-MW [65] is an architectural middleware, which supports architectural abstractions by providing implementation-level modules (e.g., classes) for representing each architectural element, with operations for creating, manipulating, and destroying the element. These abstractions enable direct mapping between a systems software architectural model and its implementation. OpenCom [21] is component-based systems-building technology for building low-level system software. It has a simple, efficient, minimal kernel, which provides a set of extension mechanisms. A subset of these extensions is the reflective extension, which provides generic support for inspecting, adapting and extending the structure and behaviour of systems at run-time. Fractal Component Model [12] is a hierarchical and reflective model for the development of component based software. A key concept in this model is membrane. Membrane plays the role of a wrapper for a Fractal component, which adds facilities beyond the functional behavior of the wrapped component. Membrane can be customized. The membrane can contain several forms of controllers each of which provides different reflective features. Life-Cycle controller facilitates reconfiguration; for instance, it provides support for starting and stopping a component.

7.1.2 Analyzing and Planning Adaptation

Self-Architecting Software SYstmes (SASSY) [63, 67] is a model-driven [6, 53] framework for composing service oriented software systems. The domain expert specifies the requirements using the Service Activity Schema language [34, 37]. With the help of a domain ontology, these requirements are translated into the system's base software architecture. After generating the base architecture, SASSY instantiates the architecture by discovering the required services and selecting the ones that maximize a global utility function that depends on the system's QoS requirements. SASSY generates alternative architectures by exploring and applying architectural patterns that increase the utility. For instance, in a situation where a service provider's availability causes the utility to be reduced, SASSY may employ a replication pattern to compose two services in a way that one can be used as a hot standby for the other. At run-time, SASSY monitors the services and computes the value of the global utility function. When it is reduced by a given threshold, SASSY re-architects the system and adapts it accordingly. SASSY uses a meta-controller [38] to configure its optimization algorithms and their parameters at run-time.

JOpera [73] provides an engine for managing the execution of service providers to satisfy multiple QoS properties, such as efficiency and latency. JOpera's objective is on enabling the service providers to satisfy their SLA requirements, while in SASSY given a set of service providers, it aims to find an architecture that satisfies the users' QoS requirements. MOSES [13] uses a BPEL specification of a software system, to dynamically determine the best composition of service providers. MOSES uses two patterns: fault-tolerant one-at a-time, referred to as sequential alternate execution of services, and fault-tolerant first-to-respond. MOSES uses linear programing to search for an optimal solution.

KAMI [43] allows for investigating how the change at run-time influence dealing with performance attributes (e.g., response time). Following a model-driven approach, KAMI uses at design time performance models based on Queuing Networks to drive architectural reasoning. It also keeps models alive at run-time through automatic re-estimation of model parameters. This way the models reflect the real behavior of the running system, and hence, the results obtained from the re-execution of the model is more accurate. In a subsequent publication [29] to KAMI, another approach is provided to address the same problem (i.e., keeping models alive at run time) by feeding a Bayesian estimator with data collected from the running system, which produces updated parameters.

Malek et al. present a framework [64] aimed at finding the most appropriate deployment architecture (allocation of software components to hardware nodes) for a distributed software system with respect to multiple, possibly conflicting QoS dimensions. For a given system, there may be many deployment architectures that provide the same functionality, but with different levels of QoS. The parameters that influence the quality of a system's deployment architecture are often not known before the system's initial deployment and may change at run-time. This means that redeployment of the software system may be necessary to improve the system's QoS properties.

7.2 Uncertainty in Self-Adaptation

A few researchers have recently begun to address uncertainty in self-adaptation. I provide an overview of those researches in the following subsections.

7.2.1 Rainbow Extension

Cheng and Garlan [16] described three specific sources of uncertainty in self-adaptation (problem-state identification, strategy selection, and strategy outcome) and provided high-level guidelines for mitigating them in Rainbow framework [42]. Problem-state identification is related to Monitoring and Analysis activities from the MAPE loop, while strategy selection and strategy outcome are related to Planning and Execution activities, respectively. In other words, they try to mitigate uncertainty in the activities of the adaptation feedback control loop.

To mitigate uncertainty in problem-state identification, they use running average in monitoring to counter variability and stochastic properties of the environment. The observations are then compared with architectural descriptions that are augmented with probabilistic information to detect trend of behavior. Once the problem is detected, an strategy is selected to resolve the problem. The uncertainty in strategy selection is mitigated by using the *Stitch* language. This language allows for modeling uncertainty (the probabilistic flavor) in strategies, therefore, when Rainbow wants to select a strategy at run-time, it can decide based on the expected value (which is capturing the uncertainty) of different strategies. Finally, once a strategy is selected and put into effect, it may succeed or fail. Instead of dealing with this uncertainty in the next adaptation loop, they consider the uncertainty in strategy outcome by specifying how long Rainbow should watch the implementation of the strategy before committing to the change. This can also be modeled using the Stitch language.

In other words, by augmenting architectural models with probabilistic models, Rainbow mitigates the uncertainty due to simplifying assumptions and noise. Moreover, by monitoring the system after adaptation Rainbow mitigates the uncertainty due to drift in the architectural models.

7.2.2 RELAX

Whittle et al. introduced RELAX [89], a formal requirements specification language that relies on Fuzzy Branching Temporal Logic to specify the uncertain requirements in self-adaptive systems. RELAX allows for explicit expression of environmental uncertainty and its effect on requirements; depending on the state of environment, RELAX specifies the requirements that can be disabled or "relaxed". To that end, RELAX introduces a set of operators that can be used in forming the requirements. These operators also define how the requirement can be relaxed at run-time. Moreover, the operators capture the kind of uncertainty (*uncertainty factor*) that can initiate the relaxation of requirements.

In a subsequent publication [15], Cheng et al. extended RELAX with goal modeling to specify the uncertainty in the objectives. They first build the goal lattice and then use it in a bottom-up fashion to look for sources of uncertainty, which are the elements of domain/environment and can endanger satisfaction of goals. In their approach, they identify uncertainty through a variation of threat modeling, which is used to identify security threats in a system. Once the uncertainty is identified, its impact is assessed to devise mitigation tactics. The ultimate tactic for mitigating uncertainty (when all other tactics fail) is to add flexibility to the goal by “relaxing” it.

7.2.3 FLAGS

Similar to RELAX, FLAGS [8] aims to achieve the basic goal of adaptive systems at the requirement level: mitigate the uncertainty associated with the environment and new business needs by embedding adaptability in the software system as early as requirement elicitation. In other words, FLAGS considers self-adaptation as a special kind of requirement, which affects other requirements. These special requirements are called adaptive goals and FLAGS allows for the definition of counter measures that must be performed if some goals are not fulfilled as expected (due to predicted uncertainty).

FLAGS also deals with another source of uncertainty in addition to the uncertainty in the context of the software: the uncertainty in the goals themselves. As satisfaction of some goals cannot be specified by simple yes–no answer, FLAGS relies on fuzzy goals for which properties are not fully known, the complete specification is not available, and small temporary violations are tolerated. Therefore, FLAGS ends up with two sets of goals: crisp goals and fuzzy goals. It formalizes the crisp goals using Linear Temporal Logic (LTL). On the other hand, it formalize fuzzy goals using its fuzzy temporal language, which is unified with the LTL specification. Therefore, all the software requirements can be specified in a single coherent language.

7.2.4 FUSION

FUSION [27], is a learning based approach to engineering self-adaptive systems. Instead of relying on static analytical models that are subject to simplifying assumptions, FUSION uses machine learning, namely Model Trees Learning (MTL) to self-tune the adaptive behavior of the system to unanticipated changes. This allows FUSION to mitigate the uncertainty associated with the change in the context of software system as it gradually learns the right adaptation behavior in the new environment. The result of learning is a set of relationships between the adaptation actions in the system and the quality attributes of interest (e.g., response time, availability). These rules consider the interaction of adaptation actions and hence to some extent mitigate the uncertainty caused due to synergy. The quality attributes of interest could be measured and collected from the running system through instrumentation of the software or sensors provided by the implementation platform [28]. The adaptation actions correspond to variation points in the software that could be exercised at run-time.

FUSION has two complementary cycles: learning cycle and adaptation cycle. The learning cycle relates the measurements of quality attributes to the adaptation actions. The learning cycle constantly monitors the environment to find possible errors in the learned relations. Persistence of such errors, which can be either due to drift or change in the context, triggers relearning the new behavior. When quality of software decreases over time and drops below a certain threshold, the adaptation cycle kicks in and uses the learned knowledge to make informed adaptation decision to improve the quality attributes. The quality of the software system is defined as aggregate collection of individual quality attributes. However, since some quality attributes may conflict with each other, the notion of utility is used to allow for making trade-offs.

7.2.5 Anticipatory Dynamic Configuration

Poladian et al. [76] studied dynamic configuration of resource-aware services, where they showed how to select an appropriate set of services to carry out a user task, and allocate resources among those services at run-time. The original work did not consider the uncertainty in the environment. Subsequently, the work was extended to make anticipatory decisions [75], and considered the inaccuracy of future resource usage predictions. To that end, they build on the previous work of one of the authors [70] and used historical profiling to find an application's resource requirements for different configurations. Considering resource availability over time mitigates the uncertainty in monitoring as it provides more accurate models of the environment being monitored.

By considering the resource availability prediction, the anticipatory model of configuration chooses a configuration that maximizes the cumulative expected value of utility over time. In other words, the selected configuration performs better in the prediction period. This reduces the number of possible future reconfigurations and in turn disruption in the system. In making the adaptation decisions, the cost of switching between the configurations is also considered; if the cost of switching is low, this approach selects a configuration that performs better at the moment and when the quality of selected configuration drops the configuration is switched. However, if the cost of switching is high, *a temporal under-optimum configuration* is accepted. That is, from the beginning an alternative configuration, which performs better over time, is selected to prevent switching later on.

7.2.6 RESIST

RESIST [19] uses information from several sources, such as monitoring internal and external software properties, changes in the structure of the software, and contextual properties to continuously furnish refined reliability predictions at run-time. The up-to-date reliability predictions express the reliability of the system in near future using probabilities. These predictions are then used to decide about changing the configuration of the software to improve its reliability in a proactive fashion. RESIST is targeted for *situated software systems*, which are prominently mobile, embedded, and pervasive. The uncertainty in these systems are prevalent as they have highly dynamic configuration, unknown operational profile/context, and fluctuating conditions, yet they are usually deployed in mission critical environments (e.g., emergency response) and have stringent reliability requirements. RESIST mitigates the uncertainty due to the context and simplifying assumptions by constant learning. Moreover, slight changes in the reliability are modeled as probability distributions indicating the noise.

RESIST takes a compositional approach to reliability estimation; the process starts with analysis at the component level, which in turn makes it possible to assess the impact of the adaptation choices on the system's reliability. The component level reliability models rely on dynamic learning techniques, specially Hidden Markov Models (HMM), to provide continuous reliability refinements. Component reliability is estimated stochastically using a Discrete Time Markov Chain and in terms of the fraction of the time spent in failure state by the component. Once the reliability of all components is obtained, a compositional model is used to determine the reliability of specific system configurations. RESIST models the uncertainty in the learning using probabilities.

Chapter 8: Formal Specification of Self-Adaptation Problem

In this chapter, I provide a formal specification of the self-adaptation problem introduced in Chapter 2, which I used to demonstrate and evaluate POISED.

8.1 Configuration

A system like the one depicted in Figure 2.1b consists of several software components, which I denote as set C. Each component $c \in C$ may have several configuration *dimensions*, which I denote as set D_c. Configuration dimensions correspond to the knobs depicted in Figure 2.1a. Each configuration dimension $d \in D_c$ may have A_d configuration *alternatives*. For example, *Video Quality* dimension of the *Camera* component in Figure 2.1 is comprised of the following alternatives: *60%*, *70%*, *80%*, *90%*, and *95%*. Configuration alternatives within the same dimension are mutually exclusive, e.g., *Video Quality* could be exactly in one of the 5 possible alternatives at any point in time.

I define the configuration space of component $c \in C$ as the Cartesian product of all the available configuration alternatives for that component:

$$ConfSpace_c \equiv \otimes_{d \in D_c} \otimes_{a \in A_d} dom(x_{c,d,a}) \tag{8.1}$$

Where x represents a decision variable with a binary domain and indicates whether an alternative has been selected or not.

A system may have several execution scenarios denoted as list $S \equiv \langle s_1, s_2, , s_n \rangle$, where each scenario $s_i \subseteq C$. A scenario represents a high-level functional capability involving the services provided by several software components. For example, the four bolded components in Figure 2.1b form a scenario dealing with the robot maneuvering a terrain. Since the same set of components could potentially form different scenarios, I represent execution scenarios as a list, instead of a set.

Configuration space of each scenario is the Cartesian product of the configuration space of all the components that participate in it:

$$ConfSpace_S \equiv \otimes_{c \in S} ConfSpace_c \tag{8.2}$$

I use $ConfSpace$ with no subscript to denote the set of all possible configurations of components in a system (i.e., $s = C$):

$$ConfSpace \equiv \otimes_{c \in C} ConfSpace_c \tag{8.3}$$

Note that some software components, in particular those that are abstractions of physical devices (e.g., camera), may not be shared among scenarios. An underlying assumption in this work is that the system has been implemented in a way that abides to such sharing constraints. Since the adaptation decisions do not impact the mapping of scenarios to components, I do not need to model such constraints. On the contrary, as detailed next, some other types of constraints need to be explicitly modeled.

8.2 Configuration Constraint

Not all configurations for a software system are valid. There may be some constraints among the system's configuration dimensions. For example, one may not be able to configure the robot in a way that *Cruise Speed* is set to *High* and *Power* is set to *Energy Saving.* Another class of constraints are among the alternatives in a given configuration dimension, i.e., the fact that configuration alternatives are mutually exclusive (recall Section 8.1). I represent these constraints as follows:

$$ConfConstraints : ConfSpace \longrightarrow \{0, 1\} \tag{8.4}$$

Where given a configuration, $ConfConstraints$ returns 0, if at least one constraint is violated, and 1, otherwise.

8.3 Quality Attribute

I use Q to denote the set of quality attributes, which are quantifiable non-functional properties of interest (e.g., response time). A quality attribute may take either discrete or continuous values, e.g., response time may take continuous values bigger than 0, while security may take an enumeration of discrete values.

The quality attributes of an execution scenario are determined by the configuration of components participating in that scenario. For example, the response time of the *Maneuver* scenario depicted in Figure 2.1b is affected by the configuration of its four components. Given a configuration of a scenario, a quality attribute is estimated via an analytical formula (model). These analytical formulas are used by the adaptation logic for making decisions. I represent an analytical formula estimating the configuration's impact on quality attribute $q \in Q$ of execution scenario $s \in S$ as:

$$\widetilde{QE}_{s,q} : ConfSpace_s \longrightarrow dom(q) \tag{8.5}$$

The tilde is the conventional notation for representing uncertainty. Description of analytical models for estimating quality attributes is beyond the scope of this paper. Numerous previous studies (e.g., [19, 27, 43, 69]) have developed analytical approaches for estimating quality attributes in terms of the system's architectural configuration. Regardless of the approach, the analytical models provide only estimates, and thus represent a source of uncertainty.

8.4 Resource

I use set R to denote the different computing resources (e.g., memory, CPU) utilized by the software system. For each resource $r \in R$, I use $\widetilde{Capacity}_r$ to represent the maximum available computing resource. While in some cases the available resource is a known constant (e.g., physical memory on a host), in other cases the available resource may fluctuate (e.g., network bandwidth), and thus introduce uncertainty.

The configuration of a system determines its resource usage. For example, consider that in the robotics system, when the *Power* dimension of the *Controller* component is configured to operate at *Full Power*, the system's battery consumption increases. I represent an analytical formula estimating the configuration's impact on the system's resource $r \in R$ as follows:

$$\widetilde{RE}_r : ConfSpace \longrightarrow dom(r) \tag{8.6}$$

The resources are different from quality attributes in two ways: (1) the user does not define preferences for their values, and (2) resources usually have certain physical limitations on their maximum usage. Numerous previous studies (e.g., [75, 76, 78]) have developed resource usage models that can be used in such setting. While sophisticated models may reduce the inaccuracy, they are not ever completely free of it, challenging the confidence with which adaptation decisions are made.

8.5 User Preference

Similar to the previous research [27,75,76,87], I use utility functions to represent the user's preferences for changes in the quality attributes. A utility function representing the user's satisfaction with quality attribute $q \in Q$ of an execution scenario $s \in S$ is represented as:

$$\widetilde{UP}_{s,q} : ran(\widetilde{QE}_{s,q}) \longrightarrow [0, 1] \tag{8.7}$$

A higher value indicates more user satisfaction with the system. Given a vector $\overrightarrow{cnf} \in ConfSpace$, I define the overall utility $\widetilde{U}$ to be the cumulative satisfaction of all the user preferences:

$$\widetilde{U} \equiv \sum_{\forall s \in S} \sum_{\forall q \in Q} \widetilde{UP}_{s,q}\left(\widetilde{QE}_{s,q}\left(\overrightarrow{cnf_s}\right)\right) \tag{8.8}$$

Where $\overrightarrow{cnf_s}$ is defined as the projection of $\overrightarrow{cnf}$ from $ConfSpace$ onto $ConfSpace_s$: $\overrightarrow{cnf_s} = proj_{\overrightarrow{I_s}}\left(\overrightarrow{cnf_s}\right)$, and $\overrightarrow{I_s}$ is the identity vector for $ConfSpace_s$. In other words, since not every component participates in every execution scenario of interest s, the above projection removes the unnecessary elements from vector $\overrightarrow{cnf}$ to derive $\overrightarrow{cnf_s}$. The assumption in the formulation of Equation 8.8 is that if the user has not specified a preference for a quality attribute of an execution scenario, the corresponding utility function returns only zero.

8.6 Optimization Problem

The objective of the adaptation logic is to find a configuration that maximizes the system's overall utility:

$$argmax_{\left(\overrightarrow{cnf} \in ConfSpace\right)} \widetilde{U} \tag{8.9}$$

The solution maximizing the above objective should satisfy two constraints. First, ensure that the solution satisfies the configuration constraints:

$$ConfConstraints\left(\overrightarrow{cnf}\right) = 1 \tag{8.10}$$

Second, ensure that the resource usage does not exceed the available resources:

$$\forall r \in R, \widetilde{RE}_r\left(\overrightarrow{cnf}\right) \leq \widetilde{Capacity}_r \tag{8.11}$$

For simplicity, the above formulation assumes the capacity of all resources is uncertain. But as you may recall from Section 8.4, this may not always be the case.

Chapter 9: Quantifying Uncertainty

Before uncertainty can be dealt with in the analysis, it needs to be quantified for a given configuration of the system. The accuracy of POISED depends on the ability to: (1) identify the sources of uncertainty, and (2) estimate the level of uncertainty. To put it boldly, my approach addresses known unknowns, not unknown unknowns. However, even if the two conditions are partially satisfied (i.e., only some sources of uncertainty are identified and estimated), POISED produces better results than the traditional approach by incorporating the known uncertainties.

In this chapter, I first describe two ways of estimating uncertainty: eliciting it from the stakeholders (e.g., user, engineer), and observing it in the system. These techniques are not intended to be exhaustive, or even generally applicable, but rather concrete examples to illustrate the feasibility of POISED. I then describe how uncertainty in the individual elements can be combined to quantify the overall uncertainty for the system. In other words, the contribution of POISED is mainly in how I use this information to reason about the uncertainty. A comprehensive framework for estimating all sources of uncertainty falls outside the scope of my thesis.

9.1 Eliciting Uncertainty from Stakeholders

Stakeholders often provide inputs for different facets of a self-adaptive system. One of the most crucial and commonly elicited inputs is the user's quality preferences. It is commonly agreed that eliciting user's preferences in terms of complex utility functions is challenging [14]. The specification of such utility functions is highly subjective and inevitably prone to uncertainty. Engineers may also provide inputs for certain software properties that cannot be easily monitored. For instance, the maximum memory consumed by a software

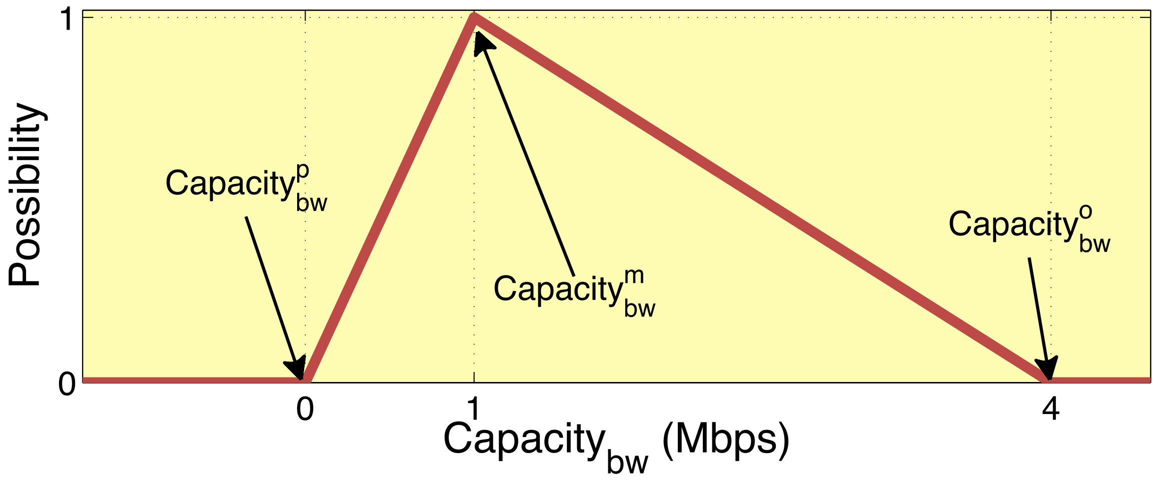


Figure 9.1: Triangular possibility distribution.

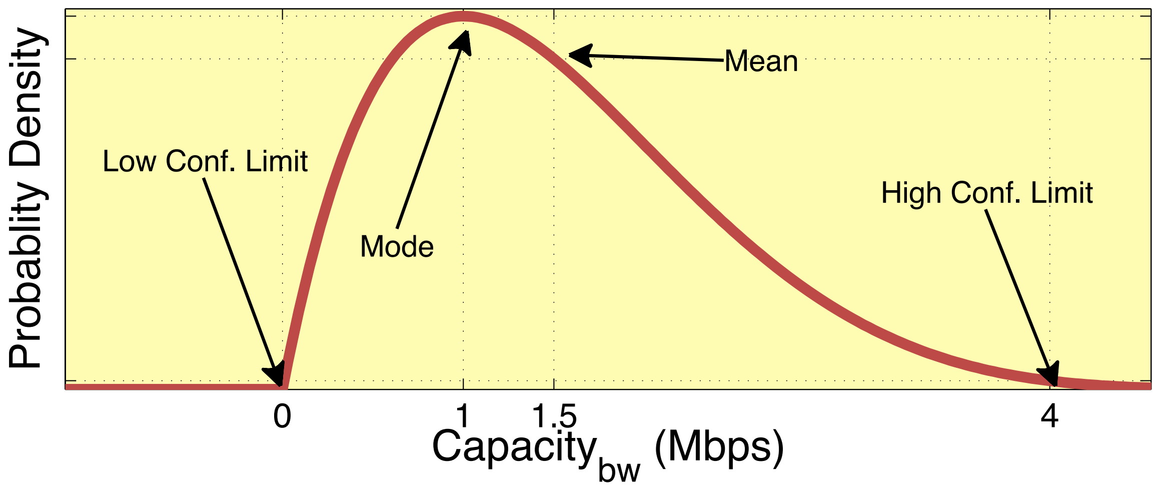


Figure 9.2: Probability distribution.

component is a property that may be available from the component's source code, but not easily obtainable through run-time monitoring. Similarly, engineers may provide inputs for certain systems parameters, such as the available network bandwidth, based on a combination of past experiences, hardware specifications, similar systems, etc.

For the inputs provided by the stakeholders, it is also reasonable to ask them to estimate the range of uncertainty based on the expected level of variation in the input. For illustration, Figure 9.1 shows how an engineer may estimate the range of uncertainty in the network bandwidth in the form of a *triangular* possibility distribution. Possibility distribution can be modeled in different ways (e.g., Gaussian, Triangular) [94], but for simplicity I use only triangular distribution in this paper. The horizontal axis marks the network bandwidth, while the vertical axis marks the possibility (i.e., degree of membership). This distribution indicates that the range of feasible values for network bandwidth is anywhere between the *most pessimistic*, denoted with $Capacity_{bw}^{p}$, and the *most optimistic*, denoted with $Capacity_{bw}^{o}$. In a triangular distribution, as we reach the boundaries, the possibility of getting the expected value drops to 0. The *most possible* value is $Capacity_{bw}^{m}$, which always has the value of 1.

In the example of Figure 9.1, the engineer sets the most possible value to be the expected network bandwidth, which may be based on the engineer's past experiences, similar systems, etc. The most pessimistic value is set to 0, representing network failure, and the most optimistic value to the ideal network bandwidth, as advertised by the network provider. By connecting the most pessimistic and optimistic points with the most possible point, we arrive at a triangular possibility distribution representing the uncertainty in the network bandwidth. A similar approach could be used for eliciting and quantifying uncertainty with the other types of stakeholder provided inputs, such as utility functions.

9.2 Measuring Uncertainty via Monitoring

A self-adaptive software system often relies on monitoring to reason about changes in the execution condition. Dynamically fluctuating system parameters are one type of phenomena in this setting that are monitored. For instance, consider that while capacity of certain resources (e.g., available physical memory) may be known prior to system's deployment, other resources (e.g., available battery charge) may change at run-time, and thus would need to be sampled during the system's execution. On top of the uncertainty created by the fluctuations in the monitored phenomenon, monitoring is also impacted by the error in the sensors used for data collection. Such an error is in particular unavoidable with digital sensors that take samples of a continuous physical phenomenon. Even if the observed phenomenon is constant, the collected data may vary, due to noisy sensors.

System parameters may not be the only elements that need to be monitored. In recent work [27, 31], I showed that analytical models used for making self-adaptation decisions are abstractions of the system and by definition make simplifying assumptions, which if not held at run-time may make the estimates inaccurate. The inaccuracy of an analytical model could be determined by comparing its estimates (i.e., $\widetilde{QE}$, $\widetilde{RE}$) against the actual behavior of the system. This can be achieved either prior to system's deployment by benchmarking the system, or through run-time observation.

The uncertainty corresponding to any monitored phenomenon can be estimated as a probability distribution. For example, if the engineer is not able to use the technique from Section 5.1 to manually specify the uncertainty in network bandwidth, an alternative approach is to estimate it by monitoring the variations and constructing the equivalent probability distribution. Figure 9.2 shows the probability distribution corresponding to the data collected for variations in the network bandwidth (i.e., $Capacity_{bw}$). There are numerous approaches for deriving a *probability density function* [44] that represents the probability distribution of collected data, including Q-Q plot [44] that could be used to estimate well-known (e.g., *normal*, *beta*) distributions, and Quantile-Regression [54] that could be used to estimate arbitrary complex distributions. In my experiments, I found Q-Q plot to be sufficient, as monitoring data often follows one of the well-known distributions. For the parameters that only have monitoring noise, I obtained approximate *normal* distribution. However, for most parameters, the distributions were skewed and best represented using *beta* distribution. As you may recall from Section 6.2, the fact that the majority of distributions were not normal was one of the primary motivations that led me to use possibilistic programming, as opposed to stochastic programming.

The majority of existing self-adaptive systems (e.g., [19,27,69,76]) ignore the probability distribution of the collected data, and simply use the mean value in their analysis. By basing the analysis on the mean behavior, they essentially ignore the statistical characteristics of the data, and thus the underlying uncertainty.

9.3 Quantifying the Overall Uncertainty

From the estimates of uncertainty in the elements comprising a self-adaptive software system, I can estimate the overall uncertainty in the system's ability to satisfy its objectives (i.e., uncertainty in the overall utility). As mentioned in Section 6.2, for efficient and effective analysis of uncertainty, I have adopted the possibilistic model of uncertainty in POISED. However, to quantify the overall uncertainty under the possibility theory, and as a fuzzy variable, I also need the uncertainty associated with each of the elements to be expressed as a fuzzy variable.

As you may recall from Section 9.1, the approach for estimating uncertainty in the inputs provided by the stakeholder already produces fuzzy variables. Recall that a possibility distribution, such as the one depicted in Figure 9.1, defines a fuzzy variable. On the other hand, I need to transform the probability distribution representing the uncertainty in monitored elements to the equivalent possibility distribution.

I demonstrate this transformation via the network bandwidth example. From the probability distribution of Figure 9.2, I can derive the corresponding possibility distribution of Figure 9.1 as follows: (1) calculate the *confidence interval* [44] and *mode* [44] of probability distribution (*mode* is the value that occurs most frequently in a distribution), (2) set the most pessimistic value equal to the *low confidence limit* of the probability distribution, (3) set the most optimistic value equal to the *high confidence limit* of the probability distribution, (4) set the most possible value equal to the mode of probability distribution, (5) connect the most pessimistic and optimistic points with the most possible point to arrive at the triangular possibility distribution of the collected data. This approach could be used to derive the possibility distribution for all of the monitored elements.

The possibility distribution derived in this way is an approximation of the probability distribution. This is because the most pessimistic and optimistic values are determined based on the confidence limits, which are not the absolute pessimistic and optimistic values for a probability distribution. While in theory the two may not be identical, in practice, selecting a large confidence level (e.g., 99%) results in negligible difference.

Using the possibility distributions quantifying the uncertainty in the individual elements of the problem, I quantify the overall uncertainty in system's ability to satisfy its overall utility (see Equation 8.8) via a possibility distribution specified as follows:

$$\begin{aligned}
U^p &\equiv \sum_{\forall s \in S} \sum_{\forall q \in Q} UP^p_{s,q}\left(QE^p_{s,q}\left(\overrightarrow{cnf_s}\right)\right) \\
U^m &\equiv \sum_{\forall s \in S} \sum_{\forall q \in Q} UP^m_{s,q}\left(QE^m_{s,q}\left(\overrightarrow{cnf_s}\right)\right) \\
U^o &\equiv \sum_{\forall s \in S} \sum_{\forall q \in Q} UP^o_{s,q}\left(QE^o_{s,q}\left(\overrightarrow{cnf_s}\right)\right)
\end{aligned} \tag{9.1}$$

Where $\overrightarrow{cnf_s} \in ConfSpace$, and U^p, U^m, and U^o denote the most pessimistic, possible, and optimistic values for the overall utility of the system, respectively. The insight is that the most pessimistic overall utility U^p occurs when all of the quality estimates QE and user preferences UP are also pessimistic. Similar insight holds for U^m and U^o. For simplicity, this formulation assumes the utility functions are monotonic, which is often the case with user-specified preferences [87]. If that is not the case, by taking the derivative of the functions I could find their extremums; details of which are elided for brevity.

Chapter 10: Possibilistic Analysis

I now describe my approach for formulating the problem of Section 8.6 as a Possibilistic Linear Programming [50] problem, and solving it using conventional solvers [1].

10.1 Possibilistic Formulation of the Problem

As you may recall from Chapter 4, POISED manages the uncertainty by finding a solution that has the best range of overall utility, where the range depends on the level of uncertainty in the system. Figure 10.1a depicts the intuition behind POISED, which involves pursuing three concurrent objectives: (1) select a configuration that maximizes $z_m \equiv U^m$, (2) minimize negative consequence of uncertainty $z_p \equiv |U^m - U^p|$, and (3) maximize positive consequence of uncertainty $z_o \equiv |U^o - U^m|$. In essence, the objective is to find a configuration that has the highest likelihood of satisfying the user's preferences given the level of uncertainty. I rewrite the objective (Equation 8.9) as a PLP as follows:

$$
\begin{aligned}
&argmin_{\left(\overrightarrow{cnf} \in ConfSpace\right)} z_p \\
&argmax_{\left(\overrightarrow{cnf} \in ConfSpace\right)} z_m \\
&argmax_{\left(\overrightarrow{cnf} \in ConfSpace\right)} z_o
\end{aligned}
\tag{10.1}
$$

Figure 10.1b shows the two possible scenarios that could occur as PLP searches for the optimal solution among three hypothetical configurations. Each configuration corresponds to a possibility distribution in Figure 10.1b, which is unlike the traditional approach, where each configuration would be a point in the utility space. The first scenario occurs when a configuration is inferior to others with respect to all objectives. For instance, in Figure 10.1b,

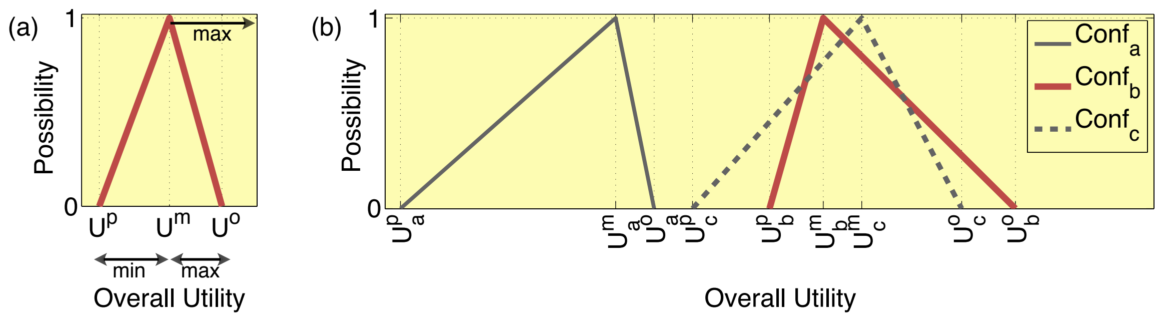


Figure 10.1: Possibilistic Linear Programming: (a) insight, and (b) trade-off between objectives.

$Conf_a$ is inferior to $Conf_b$ and $Conf_c$ with respect to all three objectives. That is $Conf_a$ has a larger z_p, and a smaller z_m and z_o. The second scenario occurs when there are trade-offs. For instance, $Conf_b$ and $Conf_c$ present a trade-off, as $Conf_b$ is superior to $Conf_c$ with respect to z_o and z_p, and inferior with respect to z_m. Section 10.2 describes the process of transforming the PLP problem to a form in which such trade-offs can be resolved.

The next step is the formulation of constraints with uncertainty, which in the problem corresponds to Equation 8.11, ensuring the resource usage does not exceed the available capacity. In Section 9.3, I described how the range of uncertainty in resource estimate and capacity can be quantified in the form of a triangular possibility distribution. Since I am dealing with a constraint, I would like to assess it under the worst case. Using the possibility distributions, I can reformulate Equation 8.11 as follows:

$$\forall r \in R, RE_r^p\left(\overrightarrow{cnf}\right) \leq Capacity_r^p \tag{10.2}$$

RE_r^p is the maximum resource usage, while $Capacity_r^p$ is the minimum resource capacity. While both RE_r^p and $Capacity_r^p$ represent the most pessimistic points in the corresponding possibility distributions, the two are semantically inverse of one another. Finally, as you may recall from Section 8.4, the capacity of certain resources may be a crisp value (e.g., available physical memory), in which case $Capacity_r^p = Capacity_r$.

10.2 Solving the Possibilistic Problem

The PLP problem is an instance of a multi-objective problem, and to solve it using commonly available linear programming solvers, I first need to transform it to an equivalent single-objective problem. This is necessary to allow me to reason about the objective trade-offs, such as those depicted in Figure 10.1b. Intuitively the transformation process entails (1) normalizing the objective functions, (2) combining the objective functions, and (3) if necessary, specifying priorities among the objectives. In this section, I describe the details of these three steps.

10.2.1 Normalizing the Objectives

Since the three PLP objectives are defined differently in terms of U, their range may not be the same. Therefore, to avoid one objective to dominate the other ones as I combine the three objectives, I first have to normalize them. I use normalizing linear membership function [58], which is a function μ that maps each objective z to a value between 0 and 1:

$$j \in \{p, m, o\}, \mu_{z_j} : dom(z_j) \longrightarrow [0, 1] \tag{10.3}$$

This allows me to have objective functions with the same range. However, for defining each function μ, I first need to determine the two extremums for each objective function z: given a configuration, the extremum maximizing the objective is called *Positive Ideal Solution (PIS)*, and the one minimizing the objective is called *Negative Ideal Solution (NIS)*. Note that the definitions of *NIS* and *PIS* are reversed when I am dealing with a minimization objective (i.e., z_p in Equation 10.1).

I can obtain these values by performing the following six single objective optimizations:

$$\begin{aligned}
z_p^{PIS} &\equiv argmin_{\left(\overrightarrow{cnf} \in ConfSpace\right)} z_p \\
z_p^{NIS} &\equiv argmax_{\left(\overrightarrow{cnf} \in ConfSpace\right)} z_p \\
z_m^{PIS} &\equiv argmax_{\left(\overrightarrow{cnf} \in ConfSpace\right)} z_m \\
z_m^{NIS} &\equiv argmin_{\left(\overrightarrow{cnf} \in ConfSpace\right)} z_m \\
z_o^{PIS} &\equiv argmax_{\left(\overrightarrow{cnf} \in ConfSpace\right)} z_o \\
z_o^{NIS} &\equiv argmin_{\left(\overrightarrow{cnf} \in ConfSpace\right)} z_o
\end{aligned} \tag{10.4}$$

I specify μ to return 1 for the *PIS* value, 0 for the *NIS* value, and proportionally linear between the two extremums:

$$\mu_{z_p} = \begin{cases} 1 & z_p < z_p^{PIS} \\ \frac{z_p^{NIS} - z_p}{z_p^{NIS} - z_p^{PIS}} & z_p^{PIS} \leq z_p \leq z_p^{NIS} \\ 0 & z_p > z_p^{NIS} \end{cases} \tag{10.5}$$

$$\mu_{z_m} = \begin{cases} 1 & z_m > z_m^{PIS} \\ \frac{z_m - z_m^{NIS}}{z_m^{PIS} - z_m^{NIS}} & z_m^{NIS} \leq z_m \leq z_p^{PIS} \\ 0 & z_m < z_m^{NIS} \end{cases} \tag{10.6}$$

Function μ_{z_o} is specified similar to μ_{z_m}. Figure 10.2 shows two instances of μ_{z_p} and μ_{z_m} that normalize the possible outputs of z_p and z_m, respectively. As the definitions of *NIS* and *PIS* are reversed for z_p, the normalizing function μ_{z_p} is decreasing, while μ_{z_m} and μ_{z_o} are increasing. This is to transform the minimization objective (i.e., z_p in Equation 10.1)

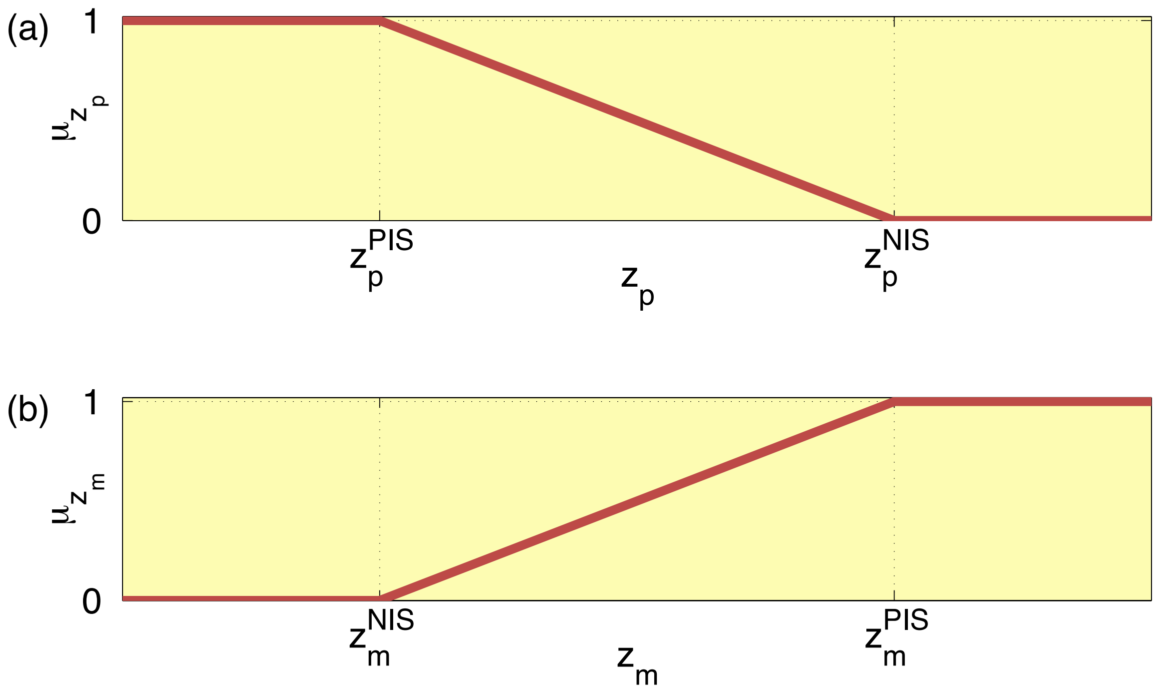


Figure 10.2: The normalizing linear membership functions: (a) the membership function for z_p, and (b) the membership function for z_m.

into a maximization one.

10.2.2 Combining the Objectives

Given the maximization objectives with the same range, I use a well-known technique[1] to specify the single-objective problem equivalent to PLP problem of Equation 10.1 as follows:

$$\begin{gathered} argmax_{\left(\overrightarrow{cnf} \in ConfSpace\right)} \varphi \\ \text{Subject to}: j \in \{p, m, o\}, \mu_{z_j} \geq \varphi \end{gathered} \tag{10.7}$$

In the above formulation, I have reformulated the objective to maximize the auxiliary

[1] *Canonical description of the mathematical technique for translating a maximization multi-objective problem to a maximization single-objective problem (see [58, 93])*: the problem of maximizing a set of objective functions $F = \{\mu_1, \mu_2, \ldots, \mu_n\}$, where all functions have the same range, is equivalent to the maximization of auxiliary variable φ, where $\forall \mu_i \in F, \mu_i \geq \varphi$.

decision variable $\varphi \in [0,1]$ representing the overall satisfaction with the three normalized objectives μ_{z_p}, μ_{z_m}, and μ_{z_o}. In other words, the auxiliary objective is to find a configuration that maximizes φ, which is constrained by the three normalized objectives, resulting first in their optimization.

10.2.3 Specifying Priorities

In the above formulation, the three objectives (expressed as constraints) have the same importance. But in certain domains, some of the objectives may have a higher priority. For instance, in a mission critical system, minimizing z_p may take precedence over maximizing z_o, since a solution capable of providing certain guarantees in the worst case scenario would be desirable. This may not be necessarily the case in other domains that are willing to tolerate higher risks with the potential of higher utility.

I achieve this by assigning weights w_p, w_m, and w_o to objectives μ_{z_p}, μ_{z_m}, and μ_{z_o}, respectively. The weights specify the important of each objective, and $w_p + w_m + w_o = 1$. Thus, the final complete optimization problem, including the constraints (Equations 8.10 and 10.2), can be formulated as follows:

$$\begin{aligned} argmax_{\left(\overrightarrow{cnf} \in ConfSpace\right)} \varphi \\ \text{Subject to} : j \in \{p, m, o\}, (1 - w_j)\, \mu_{z_j} \geq \varphi \\ ConfConstraints\left(\overrightarrow{cnf}\right) = 1 \\ \forall r \in R, RE_r^p\left(\overrightarrow{cnf}\right) \leq Capacity_r^p \end{aligned} \tag{10.8}$$

To give a normalized objective μ_{z_j} more priority over others, I make the corresponding constraint (i.e., $\mu_{z_j} \geq \varphi$) more restrictive than others. For that reason, in the above formulation, I multiply each normalized objective μ_{z_j} with $1 - w_j$, where as the value of w_j increases, the related constraint becomes more restrictive.

Chapter 11: Genetic Algorithm

Possibilistic Linear Programming can find the optimal solution. However, it is an instance of linear integer programming problem, which is know to be an NP-hard problem [20]. Adaptation decisions are usually made at run-time when timing is very important. Simply put, slow down due to scalability problems is not acceptable in such settings. Therefore, I developed an approximation optimization algorithm called genetic algorithm [22]. Genetic algorithm belongs to the class of evolutionary algorithms and can find a near-optimal solution usually in a fraction of time that takes for integer programming to find the optimal solution.

In addition to the optimization speed up due to the approximation nature of genetic algorithm, it can also benefit from parallelism to bring down the optimization time. Genetic algorithm can be designed to run on multiple processors with an insignificant overhead. Another benefit of genetic algorithm is the flexibility of models. To keep the optimization tractable, PLP puts some constraints on the models (e.g., $\widetilde{QE}$ function in Equation 8.5 should be linear). On the other hand, genetic algorithm allows for more expressive models without hampering the optimization time. Since genetic algorithm evaluates solutions programmatically, the models are as expressive as a program. In fact, it is even possible to connect genetic algorithm to a simulation engine or a queuing network model for evaluation of solutions in the optimization space (a.k.a. search space). Although this may not be advised for making adaptation decisions at run-time.

In genetic algorithm, candidate solutions are represented as *individuals*, which have a sequence of genes. The structure of the sequence, which is called *representation*, is derived from the structure of the problem and the values assigned to different genes determines the solution. Genetic algorithms start with a set of individuals (usually random) and iteratively evolves them towards ideal individuals. The set of individuals is called *population* and the population in each iteration is called a *generation*. Each individual for the next generation is produced from the current population in three steps. (1) Two or more individuals are selected from the current population stochastically. These individuals are called *parents*. It is more likely for individuals with higher quality (determined by a some *fitness* functions) to be selected as parents. (2) The new individuals, which is called *offspring*, is created based on the parents (e.g., by crossover between parents' gene sequences). (3) The new individual is slightly mutated by random modification to some of its genes.

The performance of genetic algorithm heavily depends on these three steps as well as the representation of the individuals. It is very important that the three steps carry the good characteristics of the parents to the newly created individuals. At the same time, they should prevent the search from sticking into a local optimum by accepting diversity and randomness. Finally, the individuals (i.e., sequence of genes) should be represented in a way that the chance of creating invalid individuals from valid parents is very low.

In the adaptation problem (recall Chapter 2), the representation of an individual is an array of integers. The size of the array is equal to the number of configuration knobs (recall Figure 2.1a). The genes from the parents are combined through a standard two-point crossover. In standard two-point crossover, the gene sequence is divided into three parts based on two random indexes, which are selected within the gene boundaries. The child inherits the left part (i.e., before the smaller index) and right part (i.e., after larger index) from the first parent and the middle part (i.e., between indexes) from the second parent. Finally, each gene is mutated with an independent probability of 0.05. The mutation resets the value of the gene to a valid alternative.

Figure 11.1 depicts the breeding pipeline for a configuration problem with 10 knobs. Since the problem has 10 configurations knobs, the representation of the problem also has 10 genes. In Figure, these genes are depicted as G0-G9. Here the two random indexes for two-point crossover are 2 and 7 respectively. Therefore, G0-G1 and G7-G9 are inherited from *Parent 1*, while G2-G6 are inherited from *Parent 2* to form the *Offspring*. Finally,

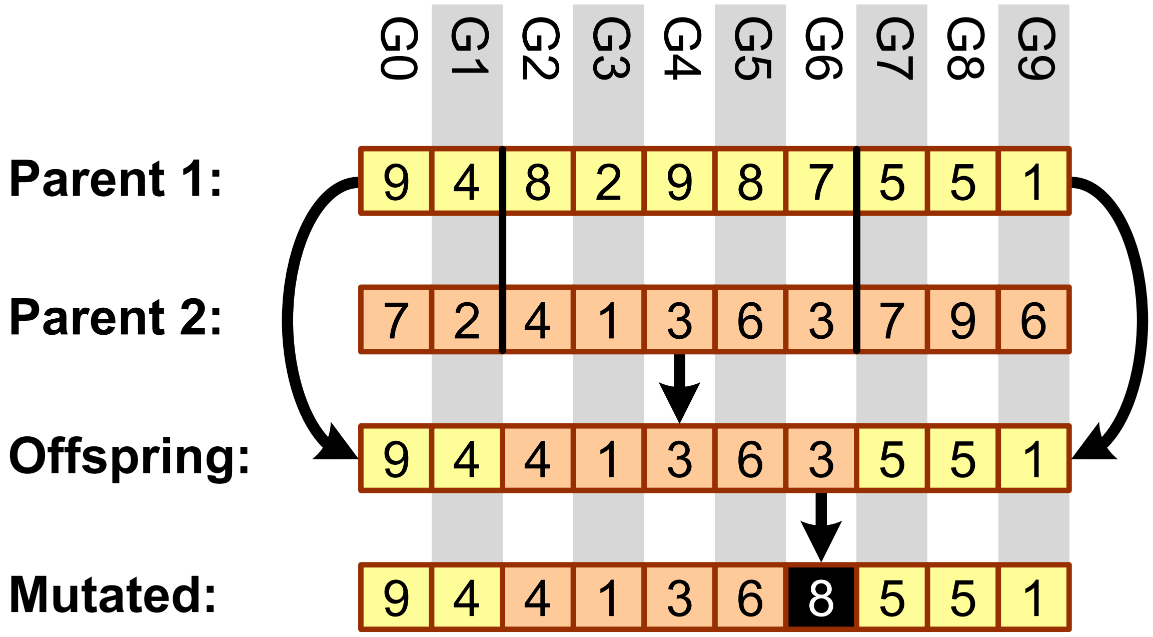


Figure 11.1: Breeding in genetic algorithm for a configuration problem with 10 knobs.

the mutation operator changes the value of G6 from 3 to 8 and the result is the *Mutated* offspring, which is added into the next generation.

To evolve populations of individuals I need to define a fitness function that evaluates the quality of each individual. The fitness function returns the minimum possible fitness if an individual is invalid (i.e., -Float.MAX_VALUE). Otherwise, it returns the value of $\widetilde{U}$ (recall Equation 8.8) for the configuration that corresponds to the individual. In the genetic algorithm, parents go through tournament selection of size 2 based on their fitness. To that end, per each parent, two random individuals are selected from the population and the individual with higher fitness value is selected as the parent. In other words, individuals with higher fitness have higher chance of passing their genetic properties to the future generations. As a result, in each iteration, the overall quality of population increases.

Modeling each configuration as an independent integer, as opposed to multiple dependent binary variables (recall Equation 8.1), prevents mutation from creating structurally invalid offspring. Moreover, the mutation operator allows offspring to diverge from the parents and slightly encourages exploration to avoid sticking into a local optimum.

The genetic algorithm stops either when the individual with absolute optimum fitness is found or when a certain limit on the search budget is met. In this case, the algorithm returns the individual, which has the highest fitness so far, as the solution and terminates the search. The genetic algorithm also notes the generation in which the search was stopped.

As I alluded to, earlier in this section, PLP has scalability problem and may not be able to converge to a solution. In this case, I am not able to evaluate quality of the solution in the genetic algorithm. Therefore, I devised a random optimization as a baseline for the genetic algorithm, which is a well-known approach for evaluation of the results in the area of evolutionary algorithms, when other baselines are not available. To that end, I select a uniformly random valid integer for each gene. This guarantees that random individuals are structurally valid. To be fair, I made the random optimization visit exactly the same number of individuals (maybe slightly more), which are visited by the genetic algorithm. This upper bound on this number can be simply calculated as $|Generation| \times |Population|$.

Chapter 12: Effecting Adaptation

Finding the ideal configuration (i.e., Chapters 10 and 11) is categorized as *Planning* according to MAPE-K loop (recall Figure 5.2). Once the ideal configuration is selected either by solving the PLP or through the genetic algorithm, the *Base-Level* software system should be changed to move from current configuration towards the ideal configuration. This is the focus of *Execution* according to MAPE-K loop.

Execution provides the necessary abstraction for Planning. While Planning can be developed abstractly and independent of Base-Level software system, Execution has close ties to the Base-Level software system. Therefore, I needed to select a target platform for implementation of Execution. I selected Prism-MW [65], which is is an *architectural middleware*, and hence, a perfect fit to my research. Note that the contributions of this work, which have been formalized in Chapter 3, belong to Planning and are abstract enough to be used in different systems.

In the remainder of this chapter I describe implementation of Execution on top of Prism-MW for effecting adaptation decisions.

12.1 Challenges with Effecting Adaptation

Typically, middleware support for architecture-based adaptation is realized in the form of adding, removing, and replacing software components. For instance, replacing a component in Prism-MW is achieved by a call to remove method followed by a call to add method on the existing architecture. However, such changes could jeopardize the integrity of a software system; they could leave the system in an inconsistent state. For instance, consider a scenario where we would like to replace the Camera component. This could be realized by removing the old Camera component and adding a new instance of it [72]. This solution,

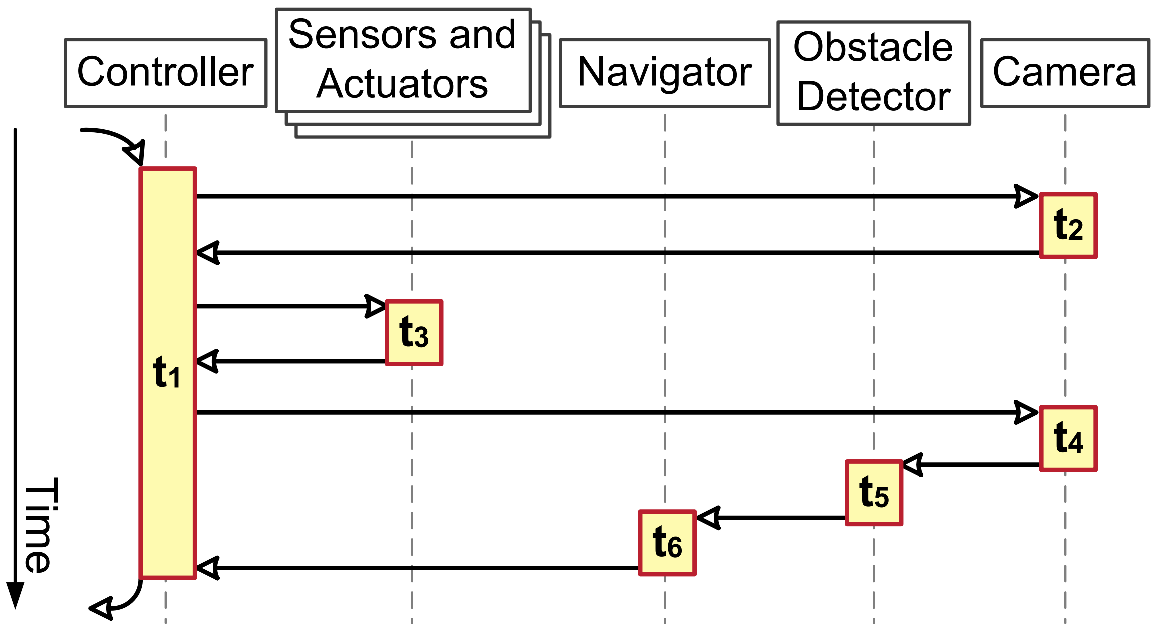


Figure 12.1: The sequence of navigation scenario and the corresponding component dependencies in EDS.

however, ignores other components (e.g., Controller) that depend on Camera for delivering their services.

Let us assume the "Navigation" scenario is according to the sequence diagram depicted in Figure 12.1. If *Camera* is replaced in the middle of such a transaction, the scenario may be start by the Controller and the old Camera and continue with the new Camera. The effect of this may manifest itself in the form of functional failure: the new Camera, which has not processed t_2 may not be able to process t_4, resulting in the system to never respond to the user (i.e., t_4, t_5, and t_6 do not occur).

At first blush it may seem that buffering events intended for the Camera component would solve the problem. However, buffering by itself cannot address consistency issues that may arise. Consider the situation in which Camera component is replaced after it has responded to t_2, but before receiving a request for t_4. In this case, it is possible for the old Map to process t_2, and the new Map to process t_4, assuming it is buffered for later processing. However, this may violate Camera's interaction protocol (i.e., t_4 can be processed only after t_2 has, which would not be the case with the newly installed Camera). Since the new Camera may not have the correct state, the system may become inconsistent.

Note that even if the component is stateless, inconsistency problems may arise. This typically happens when a stateless component provides a reverse functionality. For instance, consider a stateless encryption component that stores/retrieves data from a file using two interfaces that are reverse of one another: *cipher* and *decipher*. Replacing this component with one that uses a different type of encryption algorithm in the middle of a transaction could break the system's functionality, since decipher interface cannot be used on data that was ciphered using the old component.

By the same reasoning, in the case of stateful components, even if there is support for state transfer (i.e., ability to extract and set the component's state externally), the issue of consistency cannot be fully addressed. The reason is that the consistency of a system depends not only on the component's internal state, but also the state of its interaction with other components.

As I described in Section 7.1.1, a solution to the problem of ensuring consistency during the run-time adaptation of software was proposed by Kramer and Magee's seminal model of dynamic change management, known as quiescence [56]. In this work, a transaction is defined as an exchange of information (e.g., message, event) between two components, initiated by one of the components. Each step depicted in Figure 12.1 corresponds to a transaction. A dependent transaction is defined as a two-party transaction whose completion may depend on the completion of other consequent transactions. For instance, t_1 form a dependent transaction, while t_3 is an independent transaction.

From a theoretical perspective, the approach presented above solves the problem of ensuring consistency during adaptation. However, applying this approach in real-world software systems remains a challenge. It is typically left to the application developers to implement the required change management and coordination facilities. These facilities would provide the logic that ensures the system's consistency during adaptation (i.e., the order in which the various components are activated and passivated).

The implementation of these facilities is a major burden on the application developers for the following reasons:

1. *Identifying the component dependencies*: Determining the changes that need to occur in the system to place a software component in a particular adaptation state (i.e., active, passive) depends on the component dependencies. In event based systems, which are the focus of my work, component dependencies can be expressed in terms of transactions. Two components depend on one another, if there is a (dependent) transaction between the two. However, identifying transactions, in particular dependent transactions, requires understanding the details of the application logic (e.g., Figure 12.1), which defeats the purpose of treating components as black boxes and adapting a system at the architectural level. Identifying the dependency relationships in a large software system is very difficult.

2. *High complexity*: Realizing such facilities requires the development of complex state management and coordination logic.

3. *Lack of reuse*: Since each component has its own unique set of dependencies with other components, one component's state management logic cannot be easily reused by other software components that may need to be updated at run-time.

4. *High coupling*: Since the state management logic depends on the component dependency relationships, the resulting software is very fragile. That is as soon as the software evolves (e.g., components change the way they interact and use one another), the state management logic needs to be modified.

Traditionally, one method of reducing complexity and increasing the developer's productivity is to employ middlewares. The middleware engineers develop the frequently needed facilities (e.g., data marshalization, remote method invocation, service discovery), and provide them as reusable modules to any application that is developed on top of the middleware. Unfortunately, employing the same approach in the context of adaptation is not feasible, since the middleware designers cannot predict a priori which software components will be deployed on top of a middleware, how they will be configured, and what will be their dependencies. Therefore, modern middleware platforms do not provide change management facilities beyond simple dynamic addition and removal of components.

12.2 Using Architectural Styles to Derive Dependencies

In light of the challenges mentioned above, currently three methods of adapting a software system are employed: (1) Query the component itself to provide information about its dependency relationships. This relates to the first problem in Section 12.1, i.e., violates the black box treatment of components. Moreover, it hinders reusability of components developed in this manner. (2) Remove the old component abruptly and replace it with a new one. As exemplified using the EDS application in Section 12.1, this approach could leave the system in an inconsistent state. (3) Bring down the entire system before adapting it, and restart it afterwards. This approach results in severe disruption in system's execution. None of the existing approaches make use of advances in dynamic change management [56, 86] and hence are not desirable.

I proposed a new approach that builds on the quiescence model of dynamic change management. The key underlying insight, established in my work [33, 35] is that a software system's architectural style could reveal the dependency relationships among the components of a given system, even if the components are indirectly connected to one another. The dependency relationships are critical when adapting a software system, as they determine the impact of change on the system [56, 86].

An architectural style is a named collection of design decision, which constraint the design to a well-documented subset of possible choices [39]. Some of the most important properties of a given style are the allowable relationships among its components and connectors. If a given software system adheres to the rules and constraints of a style, I can infer the dependency relations automatically. To that end, I use the well-documented rules and constraints of a given architectural style to infer the component dependencies for the software system adhering to that style without needing additional specific information about the system.

An example of this can be seen in Figure 2.1. Since I already now the style of the robotic software system, without knowing the details of its application logic, I can derive dependency relationships between its components. For instance, since Controller depends on Camera, before doing any change to the latter, I need to make sure that the former is in a passive state. Note that this relationship is derived only from the style and topology of the architecture and without using any internal knowledge about the application. In other words, the proposed approach does not require availability of a detailed model of the component interactions in the system, such as that shown in Figure 12.1, for reasoning about quiescence.

The component dependencies are in turn used to determine a reusable sequence of steps for placing a component of a given style in the appropriate adaptation state. Such a recurring sequence of changes, which are coordinated among the system's architectural constructs (e.g., components, connectors), is called an adaptation pattern. An adaptation pattern provides a template for making changes to a software system built according to a given style without jeopardizing its consistency.

An adaptation pattern for a given style is guaranteed to be generally applicable for systems built according to that style, since (1) quiescence is guaranteed to be reachable [56], and (2) applications built according to the style must comply with the dependency relationships established in that style. I have enhanced Prism-MW with style-induced adaptation patterns discussed above. Unlike any existing solution, the middleware ensures the consistency of the adapted software system. Since each style acts according to a predefined set of rules, which are enforced by the middleware, the adaptation capability can be reused for any software built according to that style. Therefore, the middleware alleviates the application developers from implementing the same complex and error-prone functionality.

Figure 12.2 depicts the relationships between the different components in my approach.

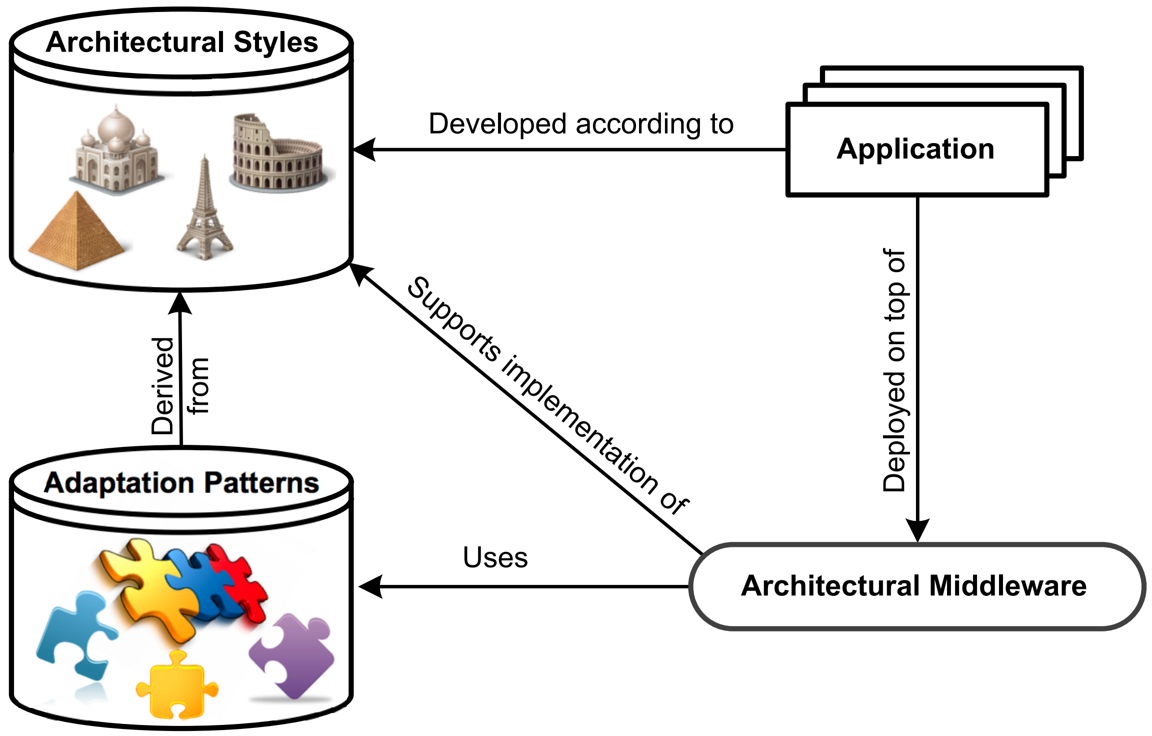


Figure 12.2: Using architectural styles to ensure safe adaptation.

An *application* is developed according to a style, which comes from a set of known *architectural styles*. For each style in this set, the related *adaptation pattern* is derived and maintained in a repository. An *adaptation pattern* is used to orchestrate the process of run-time change in the applications of the corresponding style. This way an *application* benefits from the reusable adaptation facilities provided by a *middleware* once it is deployed on top of it. Application developers can also extend the set of *architectural styles* and corresponding *adaptation patterns* to account for domain-specific styles. In the following section, I describe the derivation of adaptation patterns for two complex and well-documented architectural styles.

12.3 Style-Driven Adaptation Patterns

In this section, I describe the process of extracting adaptation patterns for two representative and complex architectural styles, C2 [82] and D3 [60]. In extracting such patterns I assume that a given component has a single style. This does not prevent a software system from having a composition of different styles as long as the components are not shared among styles. Note that while the overall approach can be applied similarly to any style, the details of the patterns, their accuracy, and level of disruption due to adaptation directly depend on the characteristics of the style. The styles with rich properties and rules inevitably result in more interesting and effective patterns.

12.3.1 Adaptation Patterns for C2 Style

During normal operation a *C2Component* receives asynchronous messages from an associated *C2Connector*. The C2 style defines how each of the architectural constructs can be connected: each *C2Component* must be connected to at least one and at most two *C2Connectors*, while a *C2Connector* can be connected to as many *C2Components* and *C2Connectors* as required. A software system built in the C2 style consists of layers, where *request* events travel upward, while *notification* events travel downward [82]. *Request* and *notification* events received from bottom and top connectors are evaluated to determine

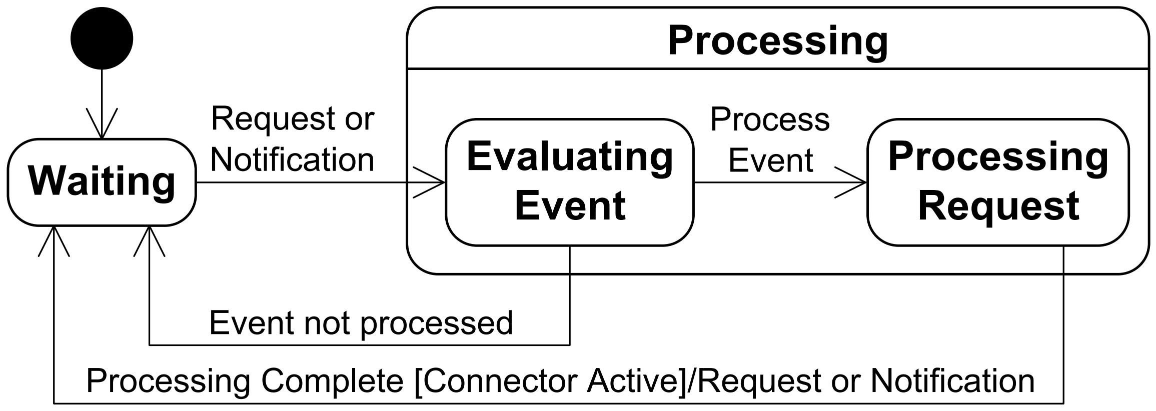


Figure 12.3: Life cycle of a C2 Component during normal operation.

which need to be processed (depicted in Figure 12.3). If the event is not intended for the component, it returns to the *Waiting* state. Otherwise, the event is processed and additional *request* and *notification* events are generated as needed. After the processing has completed and the appropriate events are sent, the component returns to the *Waiting* state.

I chose C2 style [82] due to its intended use in dynamic settings [72], and its rich set of rules and constraints that form a superset of those in simpler styles (e.g., Client-Server). Moreover, any C2 software system typically consists of many dependent transactions, making it a suitable style for describing adaptation patterns.

Adaptation of a software system requires its constituents (e.g., components, connectors) to coordinate the changes that need to occur. It is the responsibility of the adaptation module to track the adaptation state (e.g., active, passive) of the component and neighboring architectural constructs. This recurring coordination constitutes the adaptation pattern for an architectural construct in a given style.

An adaptation pattern could be expressed using statechart models (Figure 12.4). Each pattern contains one or more statecharts that define the sequence of steps a component goes through during the adaptation process. In essence, each statechart describes the run-time behavior of a component type (e.g., Client in Client-Server, Publisher in Publish-Subscribe) provisioned by a style during the adaptation process.

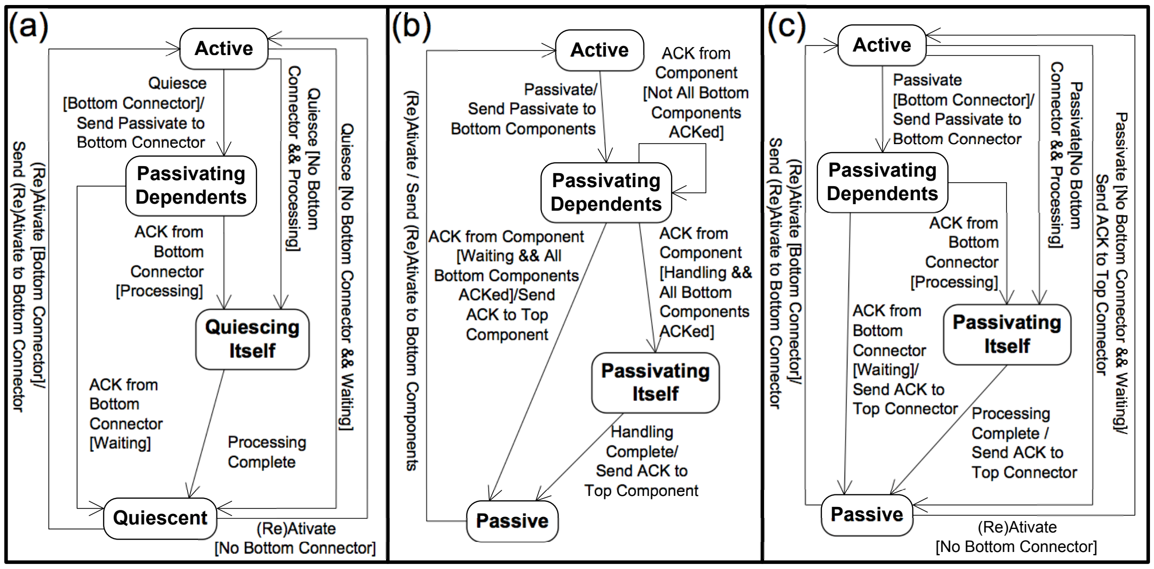

(a)
Active
Quiesce [Bottom Connector]/ Send Passivate to Bottom Connector
Quiesce [No Bottom Connector && Processing]
Quiesce [No Bottom Connector && Waiting]
(Re)Ativate [Bottom Connector]/ Send (Re)Ativate to Bottom Connector
Passivating Dependents
ACK from Bottom Connector [Processing]
Quiescing Itself
ACK from Bottom Connector [Waiting]
Processing Complete
Quiescent
(Re)Ativate [No Bottom Connector]
(b)
Active
Passivate/ Send Passivate to Bottom Components
ACK from Component [Not All Bottom Components ACKed]
(Re)Ativate / Send (Re)Ativate to Bottom Components
Passivating Dependents
ACK from Component [Waiting && All Bottom Components ACKed]/Send ACK to Top Component
ACK from Component [Handling && All Bottom Components ACKed]
Passivating Itself
Handling Complete/ Send ACK to Top Component
Passive
(c)
Active
Passivate [Bottom Connector]/ Send Passivate to Bottom Connector
Passivate[No Bottom Connector && Processing]
Passivate [No Bottom Connector && Waiting]/ Send ACK to Top Connector
(Re)Ativate [Bottom Connector]/ Send (Re)Ativate to Bottom Connector
Passivating Dependents
ACK from Bottom Connector [Processing]
ACK from Bottom Connector [Waiting]/ Send ACK to Top Connector
Passivating Itself
Processing Complete / Send ACK to Top Connector
Passive
(Re)Ativate [No Bottom Connector]

Figure 12.4: The statecharts of C2 adaptation patterns: (a) A C2Component that is being adapted, (b) A C2Connector in the substrate that depends on the adapted C2Component, and (c) A C2Component in the substrate that depends on the adapted C2Component.

The adaptation process requires a component that is to be updated to satisfy the quiescence property. The statechart in Figure 12.4a presents the transitions that take an *Active* C2Component that is being adapted to satisfy the quiescence property. When in the *Active* state, component processes received events. The first step toward quiescing the component can take one of three paths. Let us first consider the scenario where the component has no bottom connector (i.e., no other components depend on it). In this case, either the component is currently processing or waiting (idle). If the component is waiting, then it simply transitions to *Quiescent.* If the component is processing, it starts *Quiescing Itself*, and waits. When the processing has completed, it transitions to *Quiescent.*

If the component has a bottom connector (i.e., other components depend on it), then the component sends a *Passivate* request to the bottom connector to passivate the dependent components. Once an *ACK* reply is received from the bottom connector, the component transitions to either *Quiescent* if it is waiting, or to *Quiescing Itself* if it is processing. In the latter case, the component eventually transitions to the *Quiescent* when the component has completed the work. Note that according to Kramer and Magee [56] a component can only be in two states (i.e., active, passive). In the adaptation patterns, I am modeling additional states (e.g., *Quiescent*), which refer to the intermediary steps during the adaptation process, and not the state of a component.

Figure 12.4b shows the transitions that take a C2Connector in the substrate (i.e., a connector that depends on the C2Component that is currently being adapted) from *Active* to *Passive* state. The first step a connector takes is to request all of its bottom components to become passivated. Once all of the connector's bottom components have become passivated, if it is currently waiting, it transitions to the *Passive* state. Otherwise, if the connector is busy handling (routing) messages, it transitions to the *Passivating Itself* state and waits for the job to finish before transitioning to the *Passive* state.

Similar to the previous two adaptation patterns, Figure 12.4c shows the transitions that take a C2Component in the substrate (i.e., a C2Component that depends on the component currently being adapted) from *Active* to the *Passive* state.

These patterns ensure that once the component/connector transitions to the *Quiescence* state, all the components/connectors that may depend on it have also transitioned to *Passive* state. This means that the quiescence property holds for the component. By executing these patterns, the middleware waits until the component that is being adapted achieves the quiescence state before replacing it. I further detail the middleware's role in Chapter 13.

The patterns described above, while simple, codify the structural rules and constraints of C2 style into reusable logic that allows for consistent adaptation of any C2 software system. Due to space constraints I do not show the adaptation patterns of other generic styles (e.g., Publish-Subscribe, Client-Server) that I have developed so far.[1] While the overall approach of developing adaptation patterns for other styles is the same as what has been presented above, my experience suggests that different styles often result in drastically different patterns.

[1]Interested reader can access the growing repository of adaptation patterns at http://cs.gmu.edu/~smalek/AdaptationPatterns

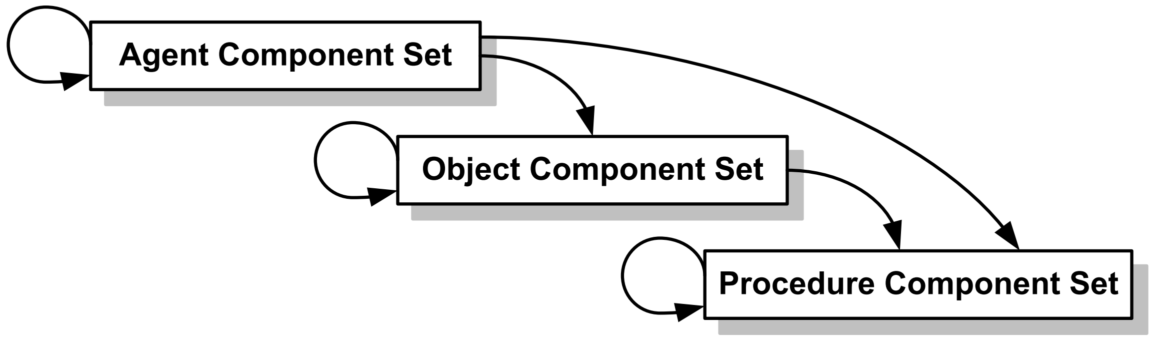


Figure 12.5: Layering between component sets in D3 reproduced from [60].

12.3.2 Adaptation Patterns for D3 Style

In the previous section I described a generic architectural style. As another example, in this section I describe a *domain-specific software architectural style*, called D3 [60]. *Domain-specific software architecture* [3, 47, 84] aims to increase productivity and reuse in the construction of software systems for a particular application domain. It captures the best practices and experiences from the domain engineers in such a way that guides the future design of similar systems [81]. An effective method of representing this knowledge is to define domain-specific component/connector types, and rules that guide their interaction, in the form of a domain-specific software architectural style.

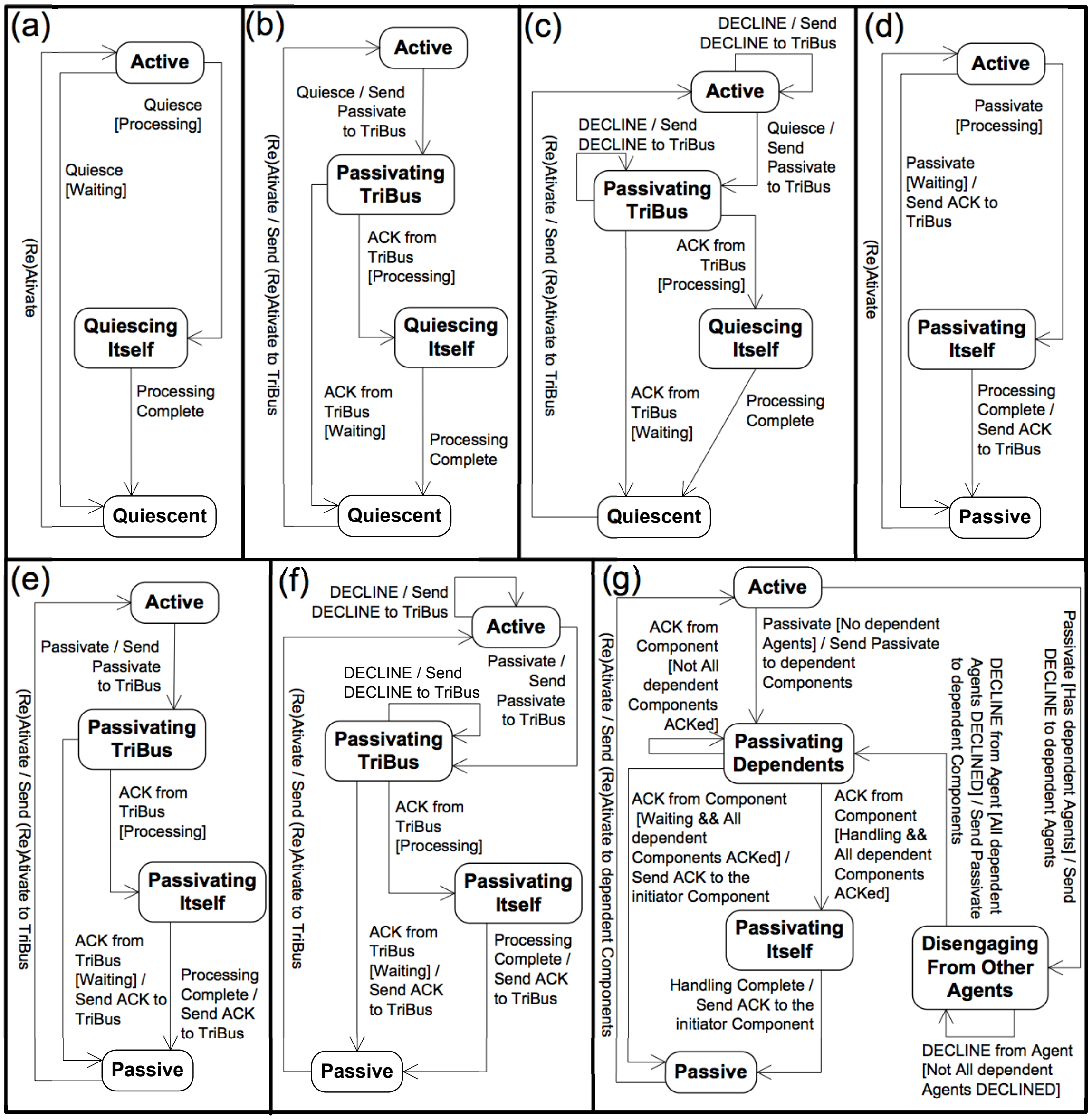
D3 [60] is an example of a domain-specific architectural style that targets the domain of Computer Supported Collaborative Design. Software systems developed for this domain facilitate concurrent construction, sharing, and synchronization of design artifacts among a distributed group of users. There are three types of components in this domain: (1) *Procedure* components are stateless functions that mainly perform commonly required transformations on the data; (2) *Object* components encapsulate data about design artifacts and the operations defined on top of them; and (3) *Agent* components are active entities which are used to help each designer to work with the system. As depicted in Figure 12.5, the components are organized in three layers of sets. In each layer a component can depend on the components that are in the layer below or at the same layer. In other words, An Agent component can access (and depend on) other Agent, Object, and Procedure components; while an Object component can only access other Object and Procedure components; and finally a Procedure component can only access other Procedure components.

The components communicate with one another via a domain-specific connector, called *TriBus*. TriBus connector provisions asynchronous request and response communication. Furthermore, the TriBus connectors enforce the rules of layering and each connection between two components should go through it. To that end, components register themselves and request connections to other components using TriBus which plays the role of mediator between two components.

The communication between components does not go beyond short term request-response relationships except communication among Agents; Agents can have long living communication sessions, which are managed using style-specific messages, such as *INFORM*, *REQUEST*, *PROPOSE*, *ACCEPT*, and *DECLINE*. An Agent can voluntarily end any prior engagement with other Agent components at any time by sending a *DECLINE* message to them. However, it should wait for receiving the *DECLINE* message back from other Agents before finishing the session.

Figure 12.6 depicts the adaptation pattern for the architectural constructs comprising D3 style. The Procedure component is one-time application of a function on data which makes the component stateless by definition. Therefore, the adaptation patterns for the Procedure component (see Figure 12.6a and d) do not need to passivate the dependent components and simple queuing of received events suffices. Note that this does not contradict with the argument about stateless components in Section 12.1, as the Procedure components are applied one-time only and do not provide a reverse functionality. On the other hand, the patterns for stateful components (see Object and Agent) need to establish the passive set first (i.e., Figure 12.6b, c, e, and f).

Extra care should be taken for managing long living communication sessions between Agents. Before adapting an Agent, all the sessions in which the Agent is present should be finished. This is achieved through sending DECLINE message to all collaborating Agents



(a)
Active
Quiesce [Processing]
Quiesce [Waiting]
(Re)Ativate
Quiescing Itself
Processing Complete
Quiescent
(b)
Active
Quiese / Send Passivate to TriBus
(Re)Ativate / Send (Re)Ativate to TriBus
Passivating TriBus
ACK from TriBus [Processing]
Quiescing Itself
ACK from TriBus [Waiting]
Processing Complete
Quiescent
(c)
DECLINE / Send DECLINE to TriBus
Active
DECLINE / Send DECLINE to TriBus
Quiesce / Send Passivate to TriBus
(Re)Ativate / Send (Re)Ativate to TriBus
Passivating TriBus
ACK from TriBus [Processing]
Quiescing Itself
ACK from TriBus [Waiting]
Processing Complete
Quiescent
(d)
Active
Passivate [Processing]
Passivate [Waiting] / Send ACK to TriBus
(Re)Ativate
Passivating Itself
Processing Complete / Send ACK to TriBus
Passive
(e)
Active
Passivate / Send Passivate to TriBus
(Re)Ativate / Send (Re)Ativate to TriBus
Passivating TriBus
ACK from TriBus [Processing]
Passivating Itself
ACK from TriBus [Waiting] / Send ACK to TriBus
Processing Complete / Send ACK to TriBus
Passive
(f)
DECLINE / Send DECLINE to TriBus
Active
DECLINE / Send DECLINE to TriBus
Passivate / Send Passivate to TriBus
(Re)Ativate / Send (Re)Ativate to TriBus
Passivating TriBus
ACK from TriBus [Processing]
Passivating Itself
ACK from TriBus [Waiting] / Send ACK to TriBus
Processing Complete / Send ACK to TriBus
Passive
(g)
Active
ACK from Component [Not All dependent Components ACKed]
Passivate [No dependent Agents] / Send Passivate to dependent Components
(Re)Ativate / Send (Re)Ativate to dependent Components
Passivate [Has dependent Agents] / Send DECLINE to dependent Agents
DECLINE from Agent [All dependent Agents DECLINED] / Send Passivate to dependent Components
Passivating Dependents
ACK from Component [Waiting && All dependent Components ACKed] / Send ACK to the initiator Component
ACK from Component [Handling && All dependent Components ACKed]
Passivating Itself
Disengaging From Other Agents
Handling Complete / Send ACK to the initiator Component
DECLINE from Agent [Not All dependent Agents DECLINED]
Passive

Figure 12.6: The statecharts of D3 adaptation patterns: (a) A Procedure that is being adapted, (b) An Object that is being adapted, (c) An Agent that is being adapted, (d) A Procedure that depends on the adapted Component, (e) An Object that depends on the adapted Component, (f) An Agent that depends on the adapted Component, and (g) The TriBus (Connector in D3 style) that is connected to the component being adapted or passivated.

(i.e., Figure 12.6c and f). The information about active sessions is encapsulated in TriBus therefore broadcasting the DECLINE message is left to adaptation pattern of TriBus (see Figure 12.6g), which is discussed further below. An Agent may receive DECLINE message from other Agents; the Agent should process the message and DECLINE back. Note that receiving DECLINE message cannot happen after the Agent has received an ACK from TriBus, as all the sessions have already finished.

Figure 12.6g shows the adaptation pattern for TriBus. Note that the pattern only manages the adaptation state related to the part of connector that deals with the connections between the component being adapted and its dependent components. In other words, the other connections established through the same TriBus are unaffected. When the TriBus receives the Passivate request from a component, it looks into its registry to find the dependent components. If the component sending the Passivate request is an Agent component and there are other Agent components depending on it, TriBus sends them the DECLINE message and waits for them to acknowledge that by sending a DECLINE message back. When that is not the case (i.e., no other Agent component is dependent on the one being adapted), the pattern passivates all the dependent components. Afterwards, the TriBus transitions into Passive state and sends an ACK to the component that initiated the process (i.e., sent Passivate request to the TriBus). As in other patterns, when the TriBus is (Re)Activated, it also (Re)Activates all of the passivated components.

Chapter 13: Implementation Framework

As I alluded to in Chapter 12, I implemented POISED on top of Prism-MW [65]. Prism-MW is an architectural middleware, which supports architecture-based development by providing implementation-level modules (e.g., classes) for representing each architectural element, with operations for creating, manipulating, and destroying the element. These abstractions enable direct mapping between a system's software architectural model and its implementation. As a result, in Prism-MW, the architectural models are completely synchronized with the implementation of the system, which alleviates the problem of architectural erosion. Prism-MW provides three key capabilities that I have relied on to realize POISED. It provides support for (1) basic architecture-level dynamism, (2) multiple architectural styles, and (3) architectural reflection.

In this section, I first provide an overview of these capabilities to familiarize the reader with this middleware. Afterwards, I provide a description of my effort for realizing Planning (recall Figure 5.2) on top of Prism-MW. Finally, I conclude this section by describing the enhancements made to Prism-MW to realize Execution (recall Figure 5.2) through style-aware adaptation of software systems. Note that I mainly relied on Prism-MW's native support for Monitoring and Analysis to complete the implementation of MAPE-K loop (recall Figure 5.2).

13.1 Overview of Prism-MW

Figure 13.1 shows the class diagram view of Prism-MW. The gray classes constitute the middleware core. *Brick* is an abstract class that represents an architectural building block. It encapsulates common features of its sub-classes (*Architecture*, *Component*, *Connector*, and *Port*). *Architecture* records the configuration of its constituent components, connectors, and

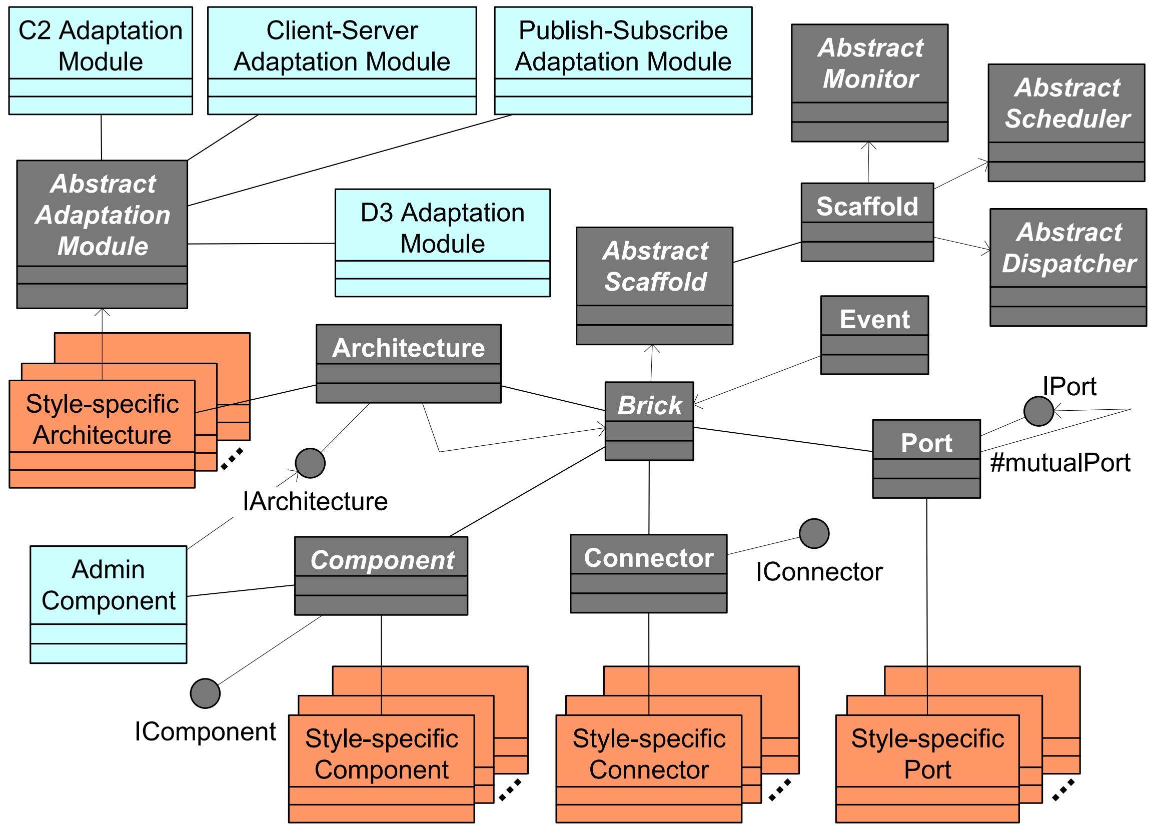


Figure 13.1: Simplified UML Class diagram of Prism-MW: middleware core shown in gray, style-specific extensions shown in orange, and enhancements for style-aware adaptation support shown in blue.

ports, and provides facilities for their addition, removal, and reconnection, possibly at system run-time. *Events* are used to capture communication in the architecture. *Components* perform computations in the architecture. The developer provides the application-specific logic by extending the component class. *Connectors* are used to control the routing of events among the attached components. Components and connectors can have an arbitrary number of attached *Ports*, which they use to attach to one another and interact. Every *Brick* in Prism-MW is associated with the *Scaffold* class, which provides a number of facilities for queuing, scheduling, monitoring, and routing of events in a distributed setting.

Prism-MW's core provides the necessary support for developing arbitrarily complex applications, as long as one relies on the provided default facilities (e.g., event scheduling, dispatching, and routing). The first step a developer takes is to sub-class from the *Component* class for all components in the architecture to implement their application-specific methods. The next step is to instantiate the *Architecture* class and to define the needed instances of components, connectors, and ports. Finally, attaching component and connector instances into a configuration is achieved by using the *weld* method of the *Architecture* class.

Through the use of the *style-specific* classes shown in orange in Figure 13.1, the middleware's default style-agnostic behavior can be modified. This feature of Prism-MW has been successfully employed to provide support for more than 20 different architectural styles, including C2, Client-Server, Publish-Subscribe, and Pipe-and-Filter [65]. For instance, a style-specific component, such as *Server*, *Publisher*, or *C2Component* can be constructed by sub-classing from the regular *Component* class and providing the style-specific logic. Similarly, as depicted in Figure 13.1, other sub-classes of *Brick* (i.e., *Architecture*, *Connector*, and *Port*) can be extended to realize the proper stylistic behavior. Interested reader may refer to [65] for more details on Prism-MW.

In Prism-MW, *Admin Component* executes the MAPE-K loop and runs the show for adaptation. It coordinates Monitoring, Analysis, and Execution, which are implemented inside Prism-MW. It also contacts implementation independent Planning module. *Admin Component* passes the Knowledge to planning Module as an XML file and receives back the optimal solution.

13.2 Finding the Optimal Solution

I have implemented Planning as an independent module, which can be integrated with Prism-MW. Planning module loosely integrates with Prism-MW through an XML file. The XML file describes the search space for Planning by enumerating the components along with all of their possible configuration. For each configuration, it specifies the impact on quality attributes. Interested reader can refer to my previous work [27, 31] on automatic assessment of impact of decisions on quality attributes.

The XML file also contains a list of constrains (recall Equation 8.4) that form the search space. The supported constraints are dependencies, conflicts, and limitations. A dependencies is used to express the situation in which a configuration from one component requires a particular configuration in another component to be selected. A conflict expresses a situation in which a configuration of one component cannot be selected when a particular configuration in another component is selected. A limitations defines a threshold on a resource usage or a particular quality attribute for overall system configuration.

Admin Component generates an XML file, which corresponds to the software system deployed on top of Prism-MW, and passes it to the Planning module. I have two parallel implementations for Planning: combinatorial and evolutionary optimization, which are based on Chapters 10 and 11 respectively. Combinatorial optimization is useful when the search space is convex and equations are linear. In this case, the optimization can converge to an optimal configuration. For large search spaces, converging to an optimal configuration may take a long time. On the other hand, evolutionary algorithm can be used to find a near optimal configuration very fast. Moreover, it does not have the limitations of the combinatorial optimization (i.e., convexity and linearity). Based on the characteristics of the search problem at hand one of these optimization algorithms can be selected.

13.2.1 Combinatorial Optimization

The combinatorial optimization is developed on top of Microsoft®Solver Foundation, which selects the best solver (e.g., Simplex algorithm [20]) based on the characteristics of the problem automatically. However, the optimization is still subject to the limitations mentioned above (i.e., convexity and linearity). Therefore, the ability to configure the optimization algorithm is very limited and the only accepted input is the XML file. Finally, since Microsoft®Solver Foundation can only run on Microsoft®Windows®, the combinatorial optimization can only be executed on a machine with Microsoft®Windows®.

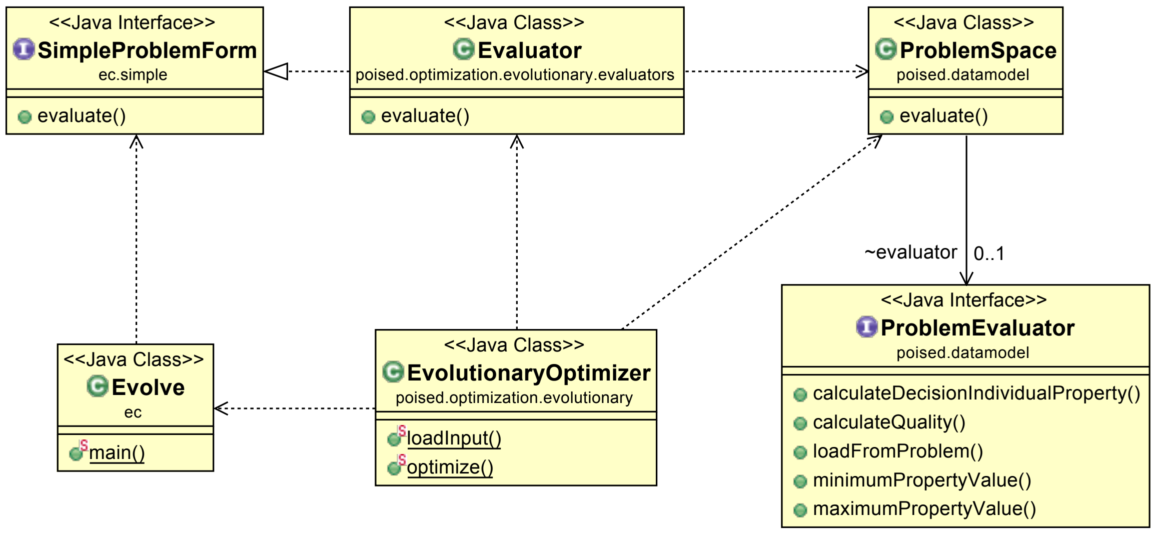


Figure 13.2: The architecture of evolutionary optimization framework, which includes ProblemEvaluator as the extension point.

13.2.2 Evolutionary Optimization

I developed an evolutionary optimization on top of ECJ [61], which is a widely used evolutionary computation framework. ECJ is based on Java, therefore, the implementation of evolutionary algorithm is portable cross-platforms. Since the evaluation of configurations (which as you recall from Chapter 11 are represented using individuals) is done programmatically, the evolutionary optimization is not faced with the limitations of the combinatorial optimization; as long as the models can be expressed using a Java program, the evolutionary optimization can use them for assessing the overall quality and suitability of the configurations. Moreover, the evolutionary optimization can find a near optimal configuration very fast, and hence, it is very scalable.

Figure 13.2 depicts the high-level architecture of the evolutionary optimization framework. The main class in the implementation is *EvolutionaryOptimizer*. It first loads the XML input through its *loadInput* method and builds in memory representation of the search space as an instance of *ProblemSpace*. *EvolutionaryOptimizer* communicates with ECJ through property files that describe the optimization process and search space.

In addition to the search space, which is expressed using the property files, ECJ requires a class to assess the overall quality and suitability of each individual during search. This class should implement the *SimpleProblemForm* interface and provide the *evaluate* method. This is the role of *Evaluator* class. However, such assessment heavily depends on the structure of the problem and system at hand. Therefore, *Evaluator* delegates this responsibility to the *ProblemSpace*.

ProblemSpace evaluates individuals using problem specific classes, which implement the *ProblemEvaluator* interface. These classes are provided by *Admin Component*. *Admin Component*, which has access to the implementation of the system on top of Prism-MW, provides both the search space (the XML file) and the problem specific evaluators (the implementation of *ProblemEvaluator* interface). Essentially, these two pieces of information form the Knowledge in MAPE-K loop (recall Figure 5.2).

Finally, once all the required information (i.e., Knowledge) is in place and loaded by *loadInput* method, *EvolutionaryOptimizer* initiates the optimization by calling the *main* method of ECJ's *Evolve* class. This is done by calling the *optimize* method in *EvolutionaryOptimizer*.

It is worth noting that the architecture of the evolutionary optimization framework (which is depicted in Figure 13.2) allows extension by providing the *ProblemEvaluator* interface. I couldn't provide this facility for the combinatorial optimization due to the limitations that I described earlier. In other words, while the implementation of evolutionary optimization is a framework that can be extended for complex systems and configured according to their needs, the combinatorial optimization is limited to the current implementation.

13.3 Style-Aware Adaptation Support

In order to support the run-time changes to the application, Prism-MW components and connectors can be added, removed, and welded during the execution of the system. As discussed further below, I have enhanced the middleware's basic dynamism with the style-driven adaptation patterns to ensure the consistency of the system during such changes.

I have realized support for the adaptation patterns in Prism-MW through three new facilities: (1) management and enforcement of the adaptation state (e.g., passive, passivating dependents, quiescence) of components and connectors, (2) realization of the adaptation patterns via several pluggable modules, and (3) execution of adaptation patterns and coordination of change management via meta-level components. I describe each of these enhancements in detail below.

13.3.1 Adaptation State

A status variable is associated with every *Brick* to determine the adaptation state of components and connectors. Recall from Figure 13.1 that both Prism-MW's *Component* and *Connector* extend the *Brick*. The *Brick* abstract class also provides *setter* and *getter* methods for accessing the status of a component and connector. Note that while some existing middleware platforms may provide information about the "liveness" property of a software component (i.e., whether or not it is active), most cannot provide additional information, such as which components are in passive mode and which ones satisfy the quiescent property. As further described below, Prism-MW tracks and updates the adaptation state of each component in the system, alleviating the component's application logic from managing its adaptation state.

In Prism-MW, *Component* class is extended for realizing the application logic. Among other services, *Component* class provides the ability to send and receive events to the application logic. I have used the same mechanism to enforce the components behaving according to their adaptation status. The *Component* class provides a wrapper that prevents "rogue" application logic from initiating a new transaction when in the Passive state. This also allows the middleware to detect the "rogue" components that ignore the passivate command and keep the active state for unlimited time. There are different ways of dealing with such components. In some domains a simple time-out mechanism [40] can solve the problem. However, in risk-averse domains informing the end-user is the best option, as this may be the sign of a problem with the component's functionality and trustworthiness.

Currently, Prism-MW could be configured to take either approach.

The third party components can also be wrapped using the *Component* abstract class in Prism-MW to enable control over them. This way Prism-MW is able to manage the component's lifecycle in the adaptation process. Prism-MW is also compatible with a third party component that can be used in different applications (e.g., services), as long as the component manages the separation between usages in different applications (e.g., state isolation) and provides a separate interface to Prism-MW for controlling the component in each usage.

13.3.2 Adaptation Module

The adaptation patterns have been realized as pluggable *Adaptation Modules*, depicted as blue classes in Figure 13.1. A style-specific *Adaptation Module* needs to override and provide an implementation for two methods inherited from *Abstract Adaptation Module*: *establish-PassiveSet* and *revive*. The first one returns the steps necessary for transitioning a component (connector) from active to passive, while the second one returns the same for achieving the reverse (i.e., from passive to active). These two methods codify state machines that correspond to the adaptation patterns, such as those described in Sections 12.3.1 and 12.3.2. Each step represents one of the transition labels in the statechart of adaptation pattern. Each step is realized as a triplet-tuple of the form ¡*condition*, *state change*, *action*¿.

One of the goals in Prism-MW, which is also something I strived for in providing the style-aware adaptation support, is the extensibility of the middleware for the needs of application developers. One of the ways that developers can extend Prism-MW is the introduction of new architectural styles [65]. My design also allows the developers to provide adaptation support for a new style by implementing the corresponding *Adaptation Module*. Thus, depending on the architectural style of a given application, different instance of *Adaptation Module* is associated with the *Architecture* object. There is no hidden feature for this as I use the same approach for providing the adaptation support for the styles currently supported in Prism-MW.

13.3.3 Admin Component

As shown in blue in Figure 13.1, I have developed a meta-level component, called *Admin Component*, which is in charge of coordinating the execution of adaptation patterns. An *Admin Component* is instantiated with every instance of the middleware and coexists with the application-level components on the same architecture container. Unlike the other components, however, it has a pointer to the *Architecture* object. Through this pointer, the *Admin Component* is capable of reflecting on both the middleware and the running application. It can identify the architectural style of the running application, use the style-specific *Adaptation Modules*, access the application's architectural constructs, and use the *setter* and *getter* methods to set their adaptation state.

Admin Component provides three interfaces: *quiesce*, *passivate*, and *activate*. They are used to place a given component in the appropriate adaptation state before/after the *Architecture*'s *add*, *remove*, and *replace* methods are invoked. Given a component to be quiesced, the *Admin Component* uses the configuration of the architecture to determine the affected components and connectors (i.e., those that need to be passivated). It also uses the *Adaptation Module*'s *establishPassiveSet* to determine the sequence of steps necessary for placing those components and connectors in passive state. Executing these steps may result in further invocations of the *Adaptation Module*'s *establishPassiveSet* on new component. *Admin Component* executes the steps by directly invoking the components using the status *setter* and *getter* methods described in Section 13.3.1. A similar process is employed to activate the components.

Chapter 14: Evaluation Results

Regarding the quality attributes, I scope the problem only to execution time. By execution time I mean how long does it take for a scenario (recall Chapter 2) to be executed in the system. I define POISED general enough so the premise of the approach can be extended to other quality attributes easily. I reduce the scope of this dissertation only to allow extensive evaluation to produce results that are statistically strong.

14.1 Experiment Setup

I have evaluated POISED on an extended version of the robotic software system (recall Chapter 2) that was developed in a previous project [62]. The robotic software used in my experiments was comprised of different software components/connectors, and different configurations. POISED treats components and connectors the same, as they are both configurable. The self-adaptation logic was tasked with satisfying 5 user preferences in terms of utility expressed as sigmoid functions; therefore, the maximum achievable overall utility (recall Equation 8.8) was 5. I used an implementation of the robotic software running on top of Prism-MW [65].

For the experiments I setup a controlled environment that allows me to create and measure the effect of uncertainty in the system. For that purpose, I controlled the execution environment of the software running on Prism-MW; I fixed the workload, and configured the software and hardware properties. I controlled the extent of random changes in the parameters. However, neither the robotic software nor POISED was controlled, which allowed them to behave as they would in practice.

The adaptation logic is a three step model: (1) Prism-MW's Admin Component generates the XML file that models the search space and passes it up to the Planning module along with the ProblemEvaluator class, (2) the Planning module finds the optimal solution, which is then passed to the Execution, and (3) the Prism-MW orchestrates the Execution based on the knowledge about the style of the software system. Since the contribution of my work is mainly in step two, I focused on evaluating that step. To make my evaluation reproducible, I captured each and every experiment in an intermediate model that describes the software system deployed on top of Prism-MW. Then I use that model to generate the XML files and feed the XML files to the Planning module. Finally, I configure the software system according to the optimal solution obtained during planning, execute it on top of Prism-MW, and collect my observations.

14.2 Quality Trade-Offs

I compared the quality of solutions selected by POISED with the traditional approach in 10 different experiments. For each experiment, I applied both approaches on the same adaptation problem. As you may recall from Chapter 4, the traditional approach is representative of the majority of existing literature that ignore the uncertainty in the adaptation decisions (i.e., base the analysis on purely crisp values obtained by calculating the mean behavior of the system properties).

I performed two types of comparison: (1) For each experiment, I compared the expected quality of solutions (configurations) selected by each approach. I refer to these results as *expected*, since they are based on the calculated consequences of uncertainty in the solutions selected by POISED. (2) I then executed the software system in the selected configuration, and observed the actual quality of solution. I refer to these results as actual, since they are based on the data collected from the system after the solution was put into effect.

I show the expected results in Figure 14.1a. The triangular possibility distribution values correspond to the solution selected by each approach. I observe a similar pattern to what was hypothesized in Figure 4.1. While POISED's solution may have a slightly lower mode [44] compared to that of the traditional approach, the overall range is always better

— POISED's most pessimistic and optimistic points are higher than that of traditional approach. This is expected, since traditional approach aims to maximize the mean behavior of the system, while POISED aims to maximize the range of behavior.

I complemented the expected ranges of the system's behavior with the actual results obtained in 30 different executions of the system under each configuration. The results are shown in Figure 14.1b. For a fair comparison, in each experiment, I fixed the application workload, as well as the range of uncertainty in the execution context. By fixing the range of uncertainty I mean controlling the range of random behavior within each source of uncertainty. Thus, different executions still resulted in different observed behaviors. I can see that the observed utilities are very closely correlated to the corresponding possibility distribution in Figure 14.1. The results show that in comparison to the traditional approach, POSIED is more likely to select a solution with better overall utility. For a meaningful comparison, in these experiments, I did not specify stringent resource constraints, which as shown next could significantly influence the outcome of both approaches. These results corroborate my first hypothesis from Chapter 3:

> **Hypothesis 1:** *Considering ranges of uncertainty instead of point estimates, in decision making, results in adaptation decisions that transition the corresponding system into a state with higher quality after adaptation.*

14.3 Performance Trade-Offs

Figure 14.2 shows the results of benchmarks comparing the execution time of POISED optimization algorithms. In addition to PLP and Genetic Algorithm (recall Chapters 10 and 11), I implemented a *Random* optimizer, which selects the optimal configuration among a set of randomly generated valid configurations. Random optimizer generates as many configurations as generated by the genetic algorithm (i.e., $|Generations| \times |Population|$) and forms the baseline for comparison among algorithms, as it depicts the case in which the proper guidance in searching the solution space is not present.

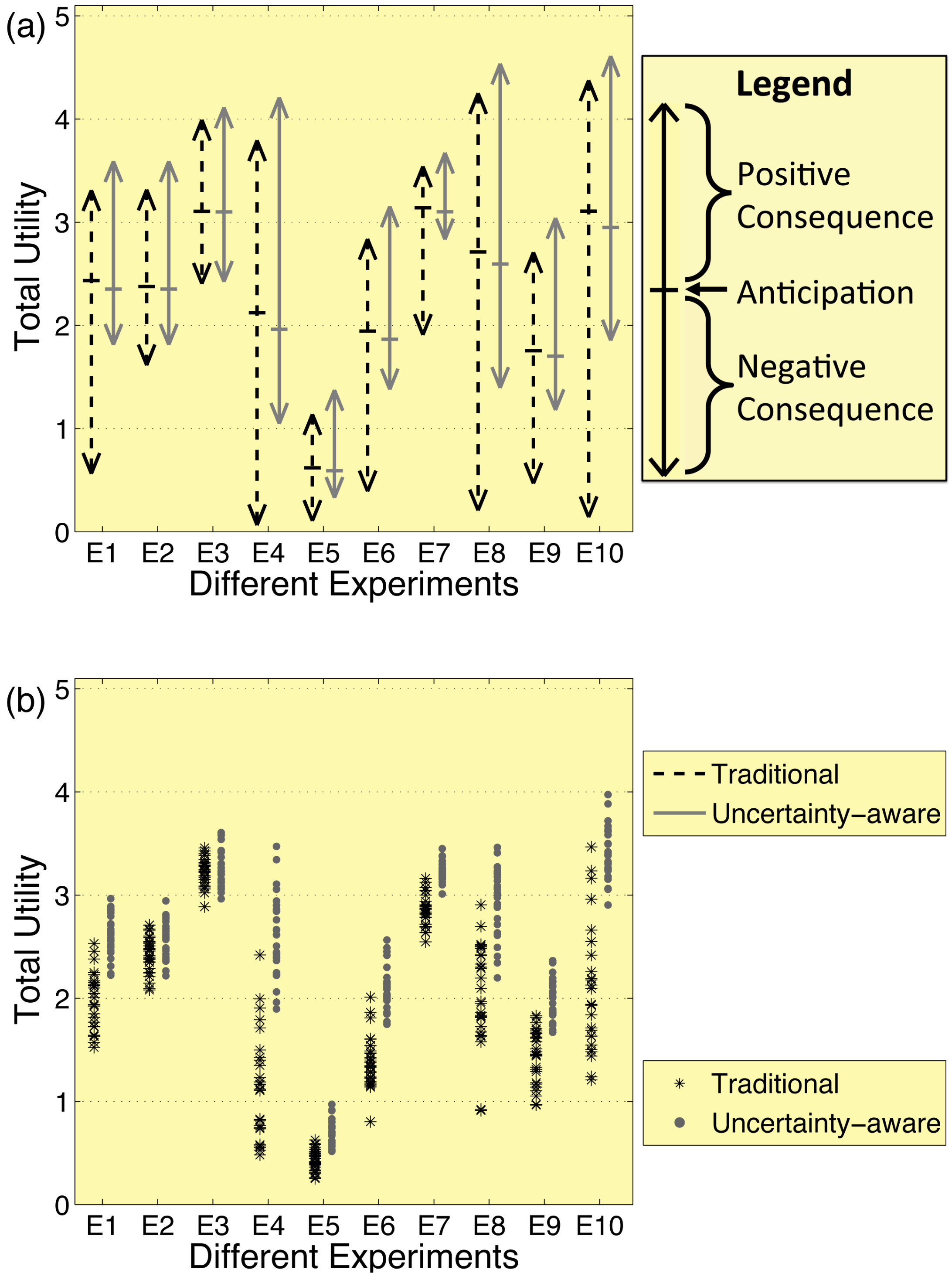


Figure 14.1: Comparison of POISED with traditional approach in 10 different experiments: (a) possibility distribution for the selected configuration, (b) 30 actual observations for each selected configuration.

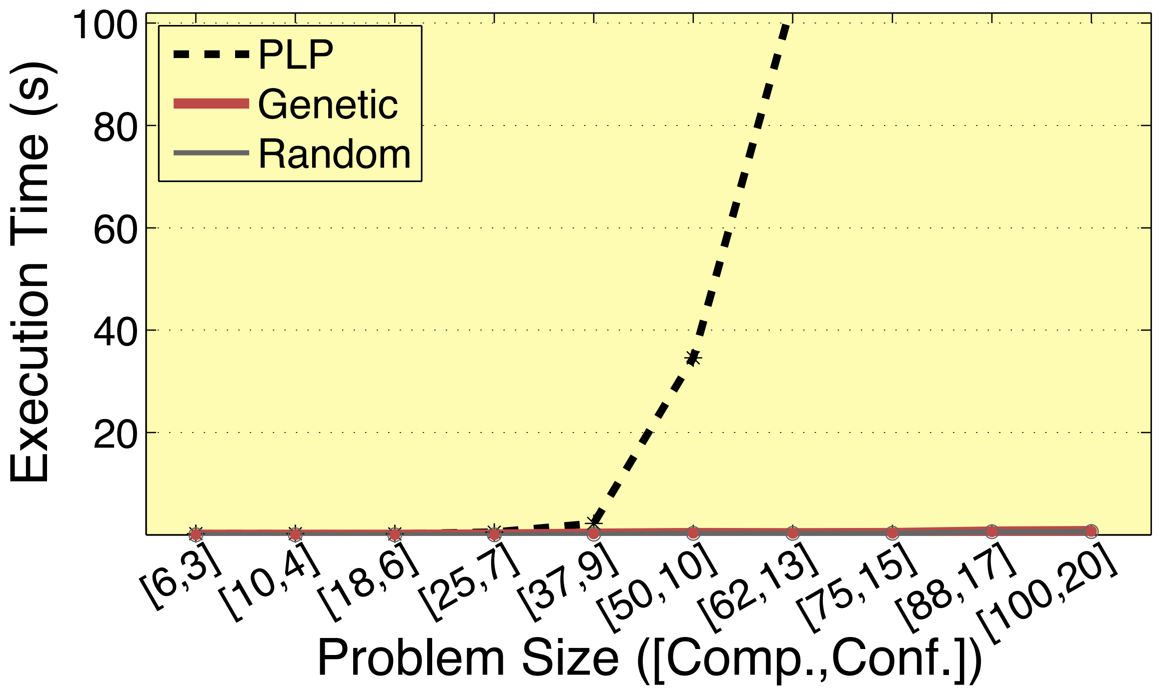


Figure 14.2: Execution time of optimization algorithms for problems of different size.

PLP is an instance of an NP-hard problem (recall Chapter 11), and hence, its execution time grows exponentially when the problem size increases. In fact, in the 3 largest problems, PLP did not converge to the optimal configuration within 30 minutes (this cut-off time is reasonable for a responsive self-adaptive software). On the other hand, the execution time of genetic algorithm and random optimizer have a very slow growth. Although it is not visible in this diagram, the execution of genetic algorithm is slightly higher that random optimizer due to bookkeeping activities in genetic algorithm. It is worth mentioning that these experiments were conducted on virtual machine running Microsoft® Windows 7™ with 2 dedicated processor cores and 2 GB of dedicated RAM to mimic resources available to a mobile robot.

Figure 14.3 shows the optimality of selected configuration by each optimization algorithm, as the other side of the coin. When the problem size is [8, 3] (i.e., 8 component with 3 different configuration alternatives for each component), the search space is very

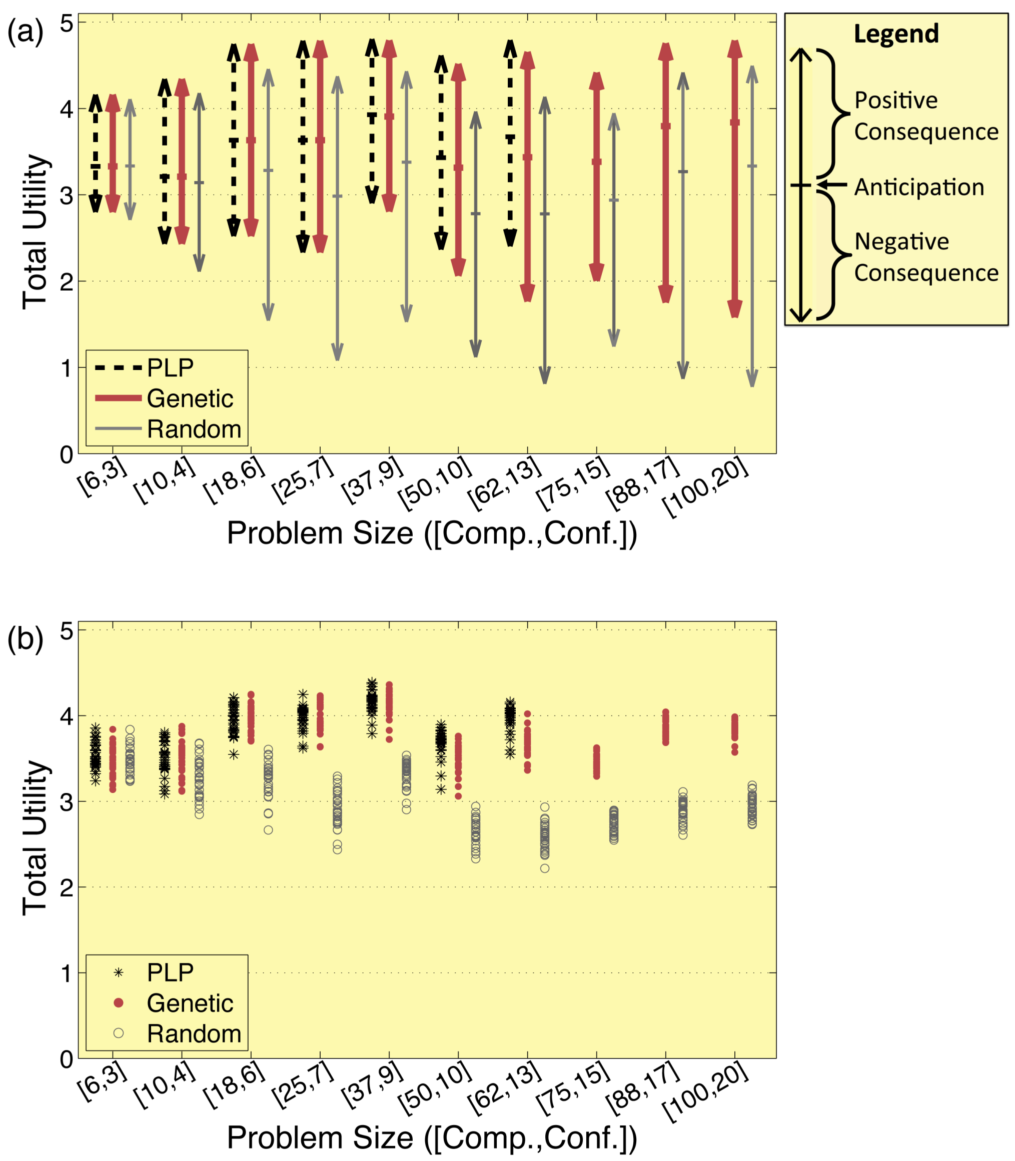


Figure 14.3: Comparison of optimization algorithms for problems of different size: (a) possibility distribution for the selected configuration, (b) 30 actual observations for each selected configuration.

small. Therefore, all three optimization algorithms are able to find the *“ideal”* configuration. However, as the size of the problems grows, the difference of algorithms both in theoretical expectation (i.e., Figure 14.3a) and actual behavior (i.e., Figure 14.3b) becomes more clear.

The quality of configurations selected by random optimizer rapidly diverges from the ideal configuration as the problems get bigger. This is due to the fact that random optimizer is not guided. In other words, it does not have any strategy for improving the configurations from one try to the next. On the other hand, genetic algorithm is a able to find a sub-optimal configuration, which is very close to ideal configuration. In fact, the difference between observation for configurations selected by genetic algorithm and PLP (which are depicted in Figure 14.3b) are not statistically significant. However, by looking at the configuration selected by genetic algorithm, I know for a fact that genetic algorithm selected a sub-optimal configuration. This verifies the effectiveness of heuristics used in genetic algorithm for guiding the search.

It took POISED longer than traditional approach to compute the optimal solution, which is not surprising, since as you may recall from Section 10.2, POISED requires 6 additional optimizations to calculate *PIS* and *NIS* values pairs for the three objective functions. However, traditional approach faces the very same dilemma; to guarantee finding the ideal configuration it should rely on an NP-hard problem, which has exponential growth. In other words, POISED is slower than traditional approach only by a constant factor.

It is true that PLP did not converge on large problems within 30 minutes in my experiments. However, it converged when the problem size was [62, 13] (i.e., 62 component with 13 different configuration alternatives for each component) in 105 seconds. This essentially entails a total of $13^{62} = 1.16 \times 10^{69}$ possible combinations. In my recent work [31], I used lower number of decisions and options to control a travel reservation system with a representative size. Therefore, although PLP has scalability limitations, it is still applicable to large problems. Moreover, in cases that the problem size is larger, one can rely on genetic algorithm to find a near optimal configuration in a fraction of time. This corroborates my second hypothesis from Chapter 3:

> **Hypothesis 2:** *Uncertainty can be incorporated in the decision-making process in a manner suitable for execution at run-time.*

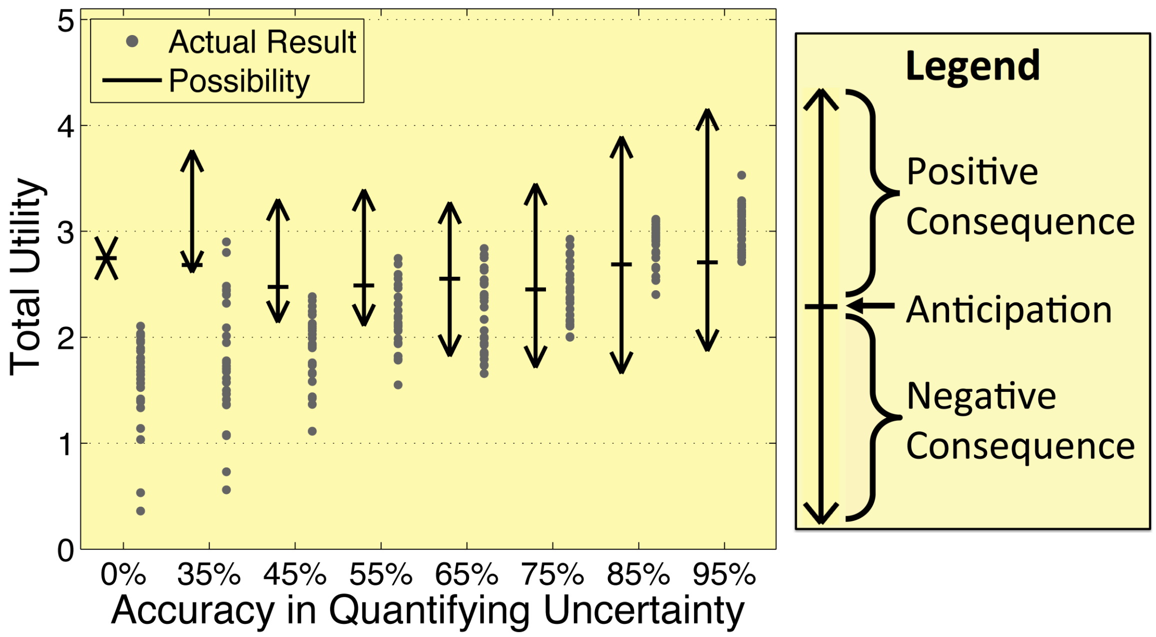


Figure 14.4: Impact of the accuracy of uncertainty estimates on the quality of POISED solutions.

14.4 Sensitivity to Uncertainty Estimates

I performed a set of experiments to evaluate how sensitive is POISED to the accuracy of uncertainty estimates. Figure 14.4 shows the results. For all of the experiments I fixed the range of uncertainty in the system parameters, as well as the workload. I changed the accuracy of uncertainty estimates used in my analysis. To that end, I simply changed the confidence level used for transforming the probability distribution corresponding to the monitored data to the equivalent possibility distribution (recall Section 9.3). As one decreases the confidence level in a probability distribution, such as the one depicted in Figure 9.2, *low confidence* and *high confidence limits* converge to *mode*, resulting in underestimation of the range of uncertainty.

The confidence levels shown on the horizontal axis of Figure 14.4 denote the accuracy of uncertainty estimates. As I decrease the confidence level from 95% to 0%, thereby making the uncertainty estimates less accurate, POISED selects configurations for which overall utilities are not borne out in practice. More specifically, since by decreasing the confidence level I underestimate the uncertainty, the actual results are lower than the expected utility. Overestimating uncertainty would have the opposite effect.

Finally, in the experiment with 0% accuracy, the most pessimistic, possible, and optimistic points are overlapping. By not considering the uncertainty, POISED is behaving the same as the traditional approach. Comparing the results of experiment with 0% accuracy to others, corroborates my third hypothesis from Chapter 3:

> **Hypothesis 3:** *Incorporating partial information about the uncertainty in analysis allows for more accurate decisions compared to having no information about uncertainty at all (i.e., considering only point estimates).*

14.5 Violation of Resource Constraints

I evaluated POISED's ability to satisfy the resource constraints under uncertainty, and compared its results against the traditional approach. I ran both approaches on the same adaptation problem but with varying levels of uncertainty in the available memory. The overall utility mode corresponding to the solution selected by each approach is shown in Figure 14.5. The robotic software system corresponding to each selected configuration was then executed 30 times. I instrumented the controlled environment to change the memory limit used by the adaptation logic (i.e., Capacity variable from Section 8.4). Parenthesized annotations in Figure 14.5 show the number of times a memory violation was observed in the actual executions of the system.

I can make several observations from these results. POISED incorporates the uncertainty in the resource usage estimates, and aims to satisfy the worst case (most pessimistic) formulation of resource constraints. Therefore, as the available memory decreases, POISED continues to select solutions that do not violate the memory constraint, but naturally have a lower utility compared to those of traditional approach. On the other hand, since the traditional approach ignores the underlying uncertainty in the estimates, as the available

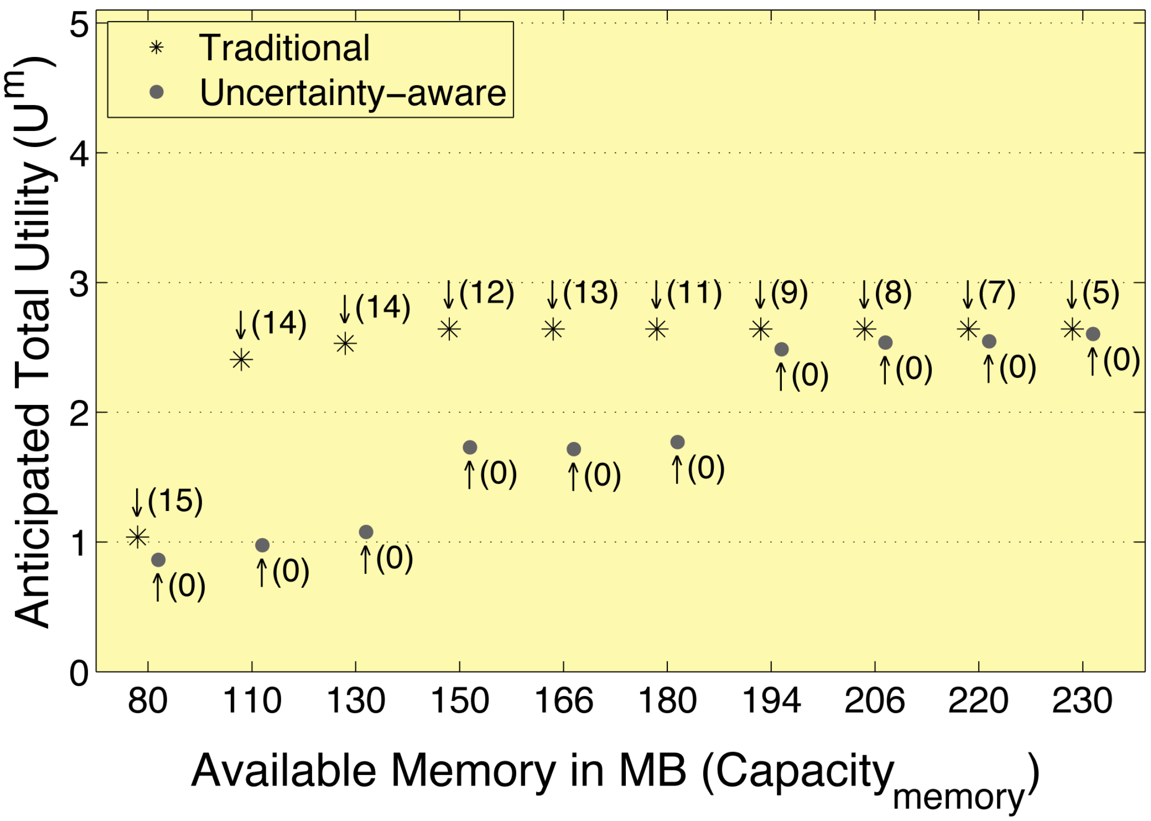


Figure 14.5: Impact of uncertainty on the overall utility and resource constraints. Number of violations is parenthesized.

memory decreases, the likelihood of selecting configurations that would violate the memory constraint increases. This pattern persists until the available memory decreases to 80MB, which is less than the mean of the memory usage estimate for configurations with high utility. The traditional approach is thus forced to select configurations with a relatively low utility. But even then, since it does not consider uncertainty, 15 executions violate the memory constraint.

14.6 Effect of Weights

In previous experiments, I placed the same weight on all objectives (i.e., $w_p = w_m = w_o = \frac{1}{3}$). But as you may recall from Section 10.2, this may not always be the case. I evaluated POISED's sensitivity to these weights on an instance of the robotic software. For a meaningful comparison, with the exception of weights, all other attributes of the system were fixed, including the range of uncertainty. Figure 14.6 shows the overall utility for the experiments. The solid bar shows the possibility distribution corresponding to the selected configuration under each weight assignment. The dots depict the observed utility for 30 executions of the software in the selected configuration. The results show the sensitivity of solutions found by POISED to the weights placed on each objective.

In the two experiments with high w_p, POISED selects a conservative solution, i.e., puts more emphasis on minimizing the negative consequence of uncertainty (z_p from Section 10.1). In the two experiments with high w_o, POISED selects a risky solution, i.e., puts more emphasis on maximizing the positive consequence of uncertainty (z_o from Section 10.1). Both approaches come at the cost of achieving mediocre overall utility mode (z_m from Section 10.1). In the two experiments with high w_m, POISED selects a solution with the best utility mode (z_m from Section 10.1), while ignoring the negative and positive consequences of uncertainty. In the last experiment, with a balanced assignment of weights, the solution achieves neither the best U^m, nor does it provide guarantees on the consequences of uncertainty. But since all of the objectives are given the same weight, it achieves

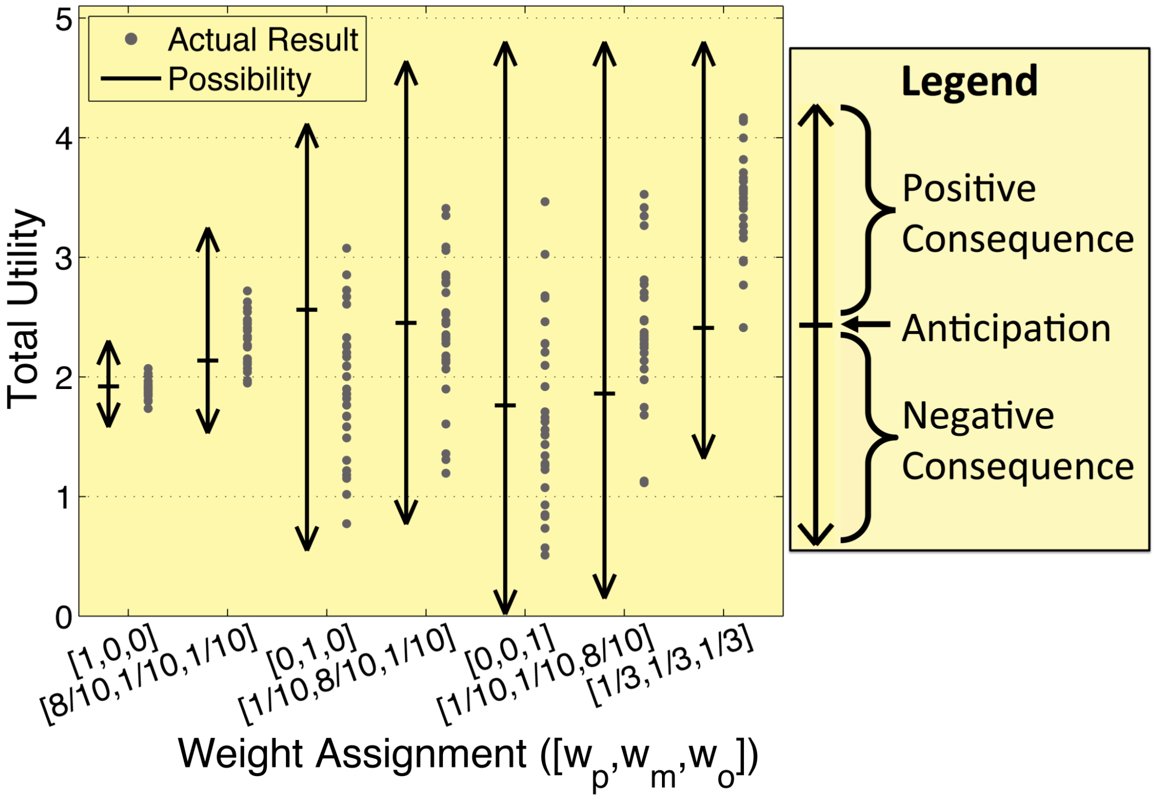


Figure 14.6: Impact of weights on the selected configuration.

a balanced set of trade-offs. However, as mentioned earlier, I can envision situations in which placing emphasis on one of the objectives may be more appropriate, which POISED allows for naturally.

Chapter 15: Design-time Decision Making

A system's early architectural decisions impact its properties (e.g., scalability, dependability) as well as stakeholder concerns (e.g., cost, time to delivery). Choices made early on are both difficult and costly to change, and thus it is paramount that the engineer gets them *"right"*. This leads to a paradox, as in early design, the engineer is often forced to make these decisions under uncertainty, i.e., not knowing the precise impact of those decisions on the various concerns. How could the engineer make the *"right"* choices in such circumstances? This is precisely the question I have tackled in my recent work [36].

In this chapter, I describe the overview of the problem of decision making at design-time. My goal is to show that I could apply the very same techniques developed in my dissertation to this new problem.

15.1 Design-time Decision Making

A software system's *early architecture* is the set of principal decisions made at the outset of a software engineering project. Early architecture encompasses choices at application, system, and hardware level that could have an impact on the software system's properties. A candidate architecture results from selecting a viable alternative for each and every decision.[1] A common practice is to carefully assess the system's early architecture for its ability to satisfy functional and non-functional requirements, as well as other stakeholder concerns, such as cost and time to delivery.

Early architectural decisions are crucial, as they determine the scope of capabilities and options that can be exercised later in the system's life cycle. Given the crucial impact of

[1]While this is a simplified way of looking at the notion of architecture, I believe it is expressive enough to capture the essence of architectural decision-making problem. Conventionally, one associates an architectural diagram as the kind of artifact an architect deals with, but in effect, such a diagram is nothing more than a collection of decisions.

early architectural decisions on the system's properties, changing them in subsequent phases of the engineering process are often both difficult and costly. At the same time, making such decisions is a complex task mired with lots of uncertainty. Getting them “wrong” poses a risk to any software engineering project.

One of the major thrusts of the software engineering research has been to transform the process of making such decisions from an art form exercised successfully by a select few to a repeatable process guided through scientific reasoning and formal analysis. A few notable examples include ATAM [17], CBAM [51], and ArchDesigner [5]. Such efforts have not aimed to replace the engineer's experience and knowledge, but to rather augment it through provisioning of appropriate methods and tools.

While great strides have been made, the existing approaches do not provide adequate support for dealing with uncertainty in early architecture [41]. In fact, there is no quantitative method of even comparing two architectures under uncertainty, let alone selecting the “right” architecture from the many possible candidates [5, 41].

In my recent work [36], I introduced GuideArch, a quantitative framework and accompanying tool aimed at guiding the exploration of architectural solution space under uncertainty. It allows the architect to make informed decisions using imperfect information. This alleviates the architect from manually sifting through an often large solution space, and instead allows her to focus on the decisions that are critical to the system's success.

Unlike any existing approach [5, 17, 51], GuideArch explicitly represents the inherent uncertainty in the knowledge and incorporates that in the analysis. It enables an incremental method of making and refining architectural decisions throughout the engineering process. As the rough estimates in the early stages give way to precise estimates in the later stages, GuideArch allows the architect to refine the models and explore other suitable alternatives.

GuideArch employs *fuzzy mathematical* methods [94] to reason about uncertainty. I used the very framework introduced in this dissertation to quantitatively compare architectural candidates under uncertainty.

Table 15.1: Overview of SAS Case Study.

Decisions▼	Alternatives▼
Location Finding	1: GPS
	2: Radio triangulation
File Sharing Package	1: OpenIntents
	2: In house
Report Synchronization	1: Explicit
	2: Implicit
Chat Protocol	1: XMPP (Open Fire)
	2: In house
Map Access	1: On demand (Google)
	2: Cached on Server
	3: Preloaded (ESRI)
Hardware Platform	1: Nexus 1 (HTC)
	2: Droid (Motorola)
Connectivity	1: Wi-Fi
	2: 3G on Nexus 1
	3: 3G on Droid
	4: Bluetooth
Database	1: MySQL
	2: sqlLite
Architectural Pattern	1: Facade
	2: Peer-to-peer
	3: Push-based
Data Exchange format	1: XML
	2: Compressed XML
	3: Unformatted data

Properties▼
Battery Usage
Response Time
Reliability
Ramp up Time
Cost
Development Time
Deployment Time

15.2 Research Objective

I used a subset of EDS, called *Situational Awareness System* (*SAS*), to motivate, describe, and evaluate GuideArch. SAS is a mobile software system and is developed in collaboration with a government agency for the deployment of personnel in emergency response scenarios. SAS is intended to allow the emergency crew carrying Android devices to share and obtain an assessment of the situation in real-time (e.g., interactive overlay on maps), and coordinate with one another (e.g., send reports, chat, and share video streams).

In architecting the SAS application, a team, consisting of academics and engineers from

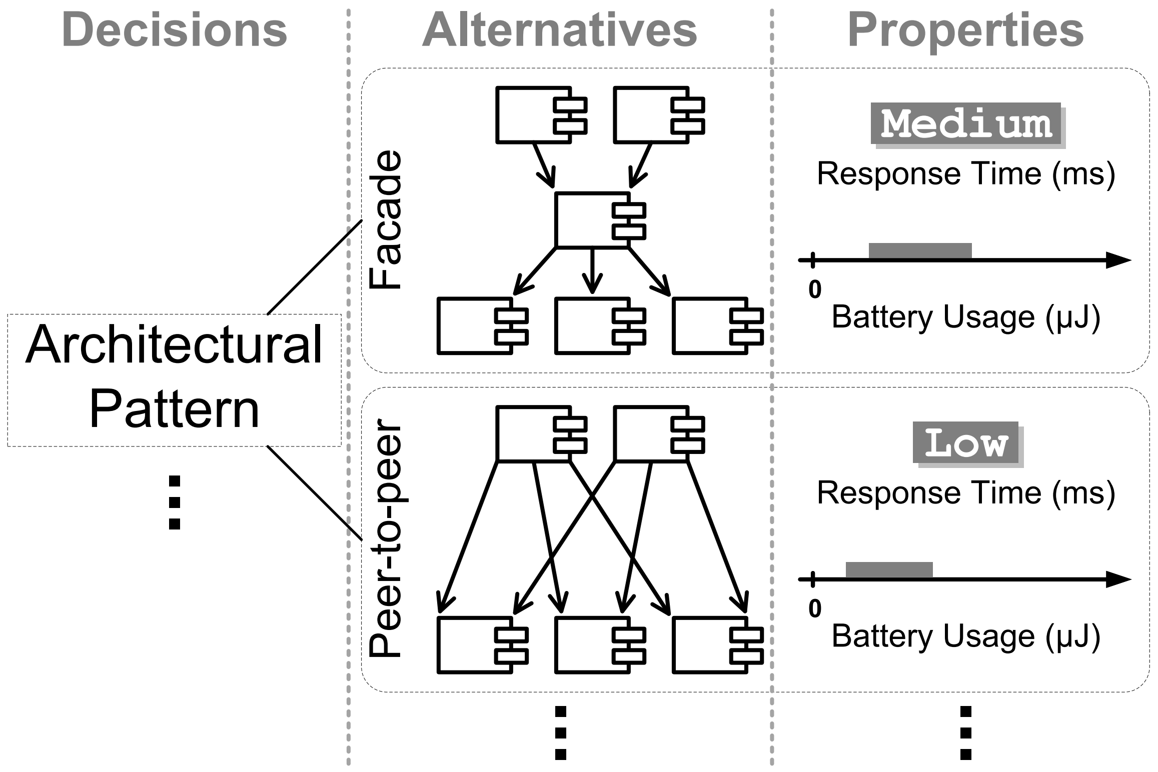


Figure 15.1: Architectural Pattern decision along with its alternatives and their impact on the properties in SAS's early architecture.

the agency, was formed to decide among the early design decisions. Table 15.1 shows the decisions and alternatives that comprised the SAS project. The requirements posed by the entities within the agency sponsoring the project also called for several areas of concern, which the team derived over several project meetings with the various stakeholders. The concerns are referred to as properties of the architecture/system. Although not depicted in the table, some of the decisions depend on one another. For instance, the choice of *Connectivity* depended on the selected *Hardware Platform*.

Figure 15.1 illustrates the relationship between decisions, alternatives, and properties. Each decision consists of several viable alternatives, the selection of which results in different candidate architectures. Each architecture in turn exhibit its own unique properties, e.g., *Response Time* and *Battery Usage* in Figure 15.1 are two such properties. My experiences with SAS and other systems show that precisely predicting the impact of an architectural alternative on the system's properties is extremely difficult, particularly in early phases of engineering. This difficulty is due to uncertainty.

Previous approaches targeted at early architecting [5, 17, 51] have ignored uncertainty, and assumed the impact of early architectural decisions can be quantified precisely. I collectively refer to them as the *traditional approaches.* A body of literature (e.g., [9,23,66]) has investigated the use of probabilistic models for representing uncertainty in the system's architecture, but such approaches are only useful in settings where probabilistic information is available (e.g., at run-time), which is often not the case in the early architecture phase.

The challenge motivating GuideArch is how to enable the architect make informed decisions in such circumstances? Answering this question requires us to first understand how the practitioners make these decisions. Although an architect may not be able to precisely quantify the impact of decisions on the system's properties, surely she must have some intuition or rough estimate of their impact, as no successful system comes to existence through random decisions [83]. This intuition may be based on the data available from similar designs in other systems, architect's prior knowledge, manufacturer specification, scientific publication, expert judgment, etc. For instance, as depicted in Figure 15.1, when presented with two alternative architectural patterns for the system, the architect may be able to distinguish between the two by classifying one's impact on *Response Time* to be *Low*, and the other one as *Medium.* In some other cases, the architect may have more specific knowledge of the situation and specify the impact as a range of values. In the example of Figure 15.1, the architect specifies the impact on *Battery Usage* as a range. GuideArch is a novel quantitative framework and supporting tool that given such loosely defined estimates help the architect with making the best decisions.

15.3 Uncertainty in Design

Before delving into the approach, it is important to clarify what I mean by uncertainty. The scope of uncertainty here has to do with not knowing the exact impact of architectural alternatives on properties of interest, i.e., not being able to precisely specify the impact as

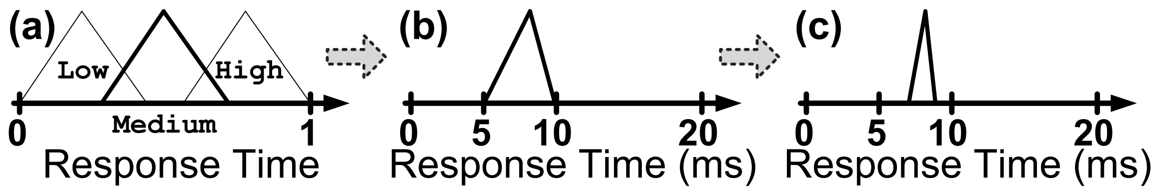


Figure 15.2: Evolution of how the architect expresses impact on properties: (a) enumerated values, where the bold one represents the one that is selected, (b) range of concrete values, and (c) convergence towards a crisp value.

a crisp value. However, there are other sources of uncertainty in early architecting that are not tackled in my work. Consider for instance the uncertainty introduced by the following questions: Have all of the properties of concern been elicited? Have all of the decisions and alternatives been identified? [71] While the ability to answer such questions is clearly crucial, they fall outside the scope of my research.

As alluded to in Section 15.2, the architect may estimate the impact of an alternative on the properties in two ways: (1) *Range:* When the architect has a rough estimate of the value, the architect could specify it as a range, e.g., a particular choice is anticipated to have $10\mu J$ of battery usage, with $8\mu J$ and $14\mu J$ in optimistic and pessimistic situations, respectively. (2) *Enumeration:* Sometimes the architect may have no way of quantifying the impact, even as a range. In such cases, the architect abstractly specifies the impact of an alternative in relation to other alternatives through enumeration, e.g., certain alternative is likely to have a *Low* response time. Here, *"Low"* may not say much about the actual values (e.g., seconds), but nevertheless carries useful information that influences the decisions early on.

These two mechanisms are aligned with the way humans in general conceptualize uncertainty and provide an intuitive method of modeling the architect's imperfect knowledge. The key contribution of GuideArch is the ability to provide quantitative analysis of the trade-offs given such loose specifications. I achieve this by representing the uncertainty as a *fuzzy value.*

I take slightly different approaches to transform the enumerated values and range of values to the corresponding fuzzy representations. As shown in Figure 15.2a, the enumerated value is simply mapped to fuzzy values that divide the normalized domain equally. I do this because the architect has no way of quantifying the impact of alternatives on the properties, yet has some intuitions as to how alternatives compare to one another. On the other hand, when the architect can specify the impact as a range (Figure 15.2b), I assign the possibility of 1 to the *anticipated* (most likely) value, and possibility of 0 to the *optimistic* and *pessimistic*, respectively. I let the possibility to decrease linearly from the anticipated to the optimistic and pessimistic points.

As a software project progresses, the architectural models become more concrete and enriched with information collected from simulations and prototypes, subsequently increasing the accuracy with which impact of alternatives can be estimated. Thus, as depicted in Figure 15.2, I expect to see the impact of alternatives expressed mostly through enumeration at the outset of a project to give way to ranges of values in later iterations, and eventually converge to crisp values toward the end.

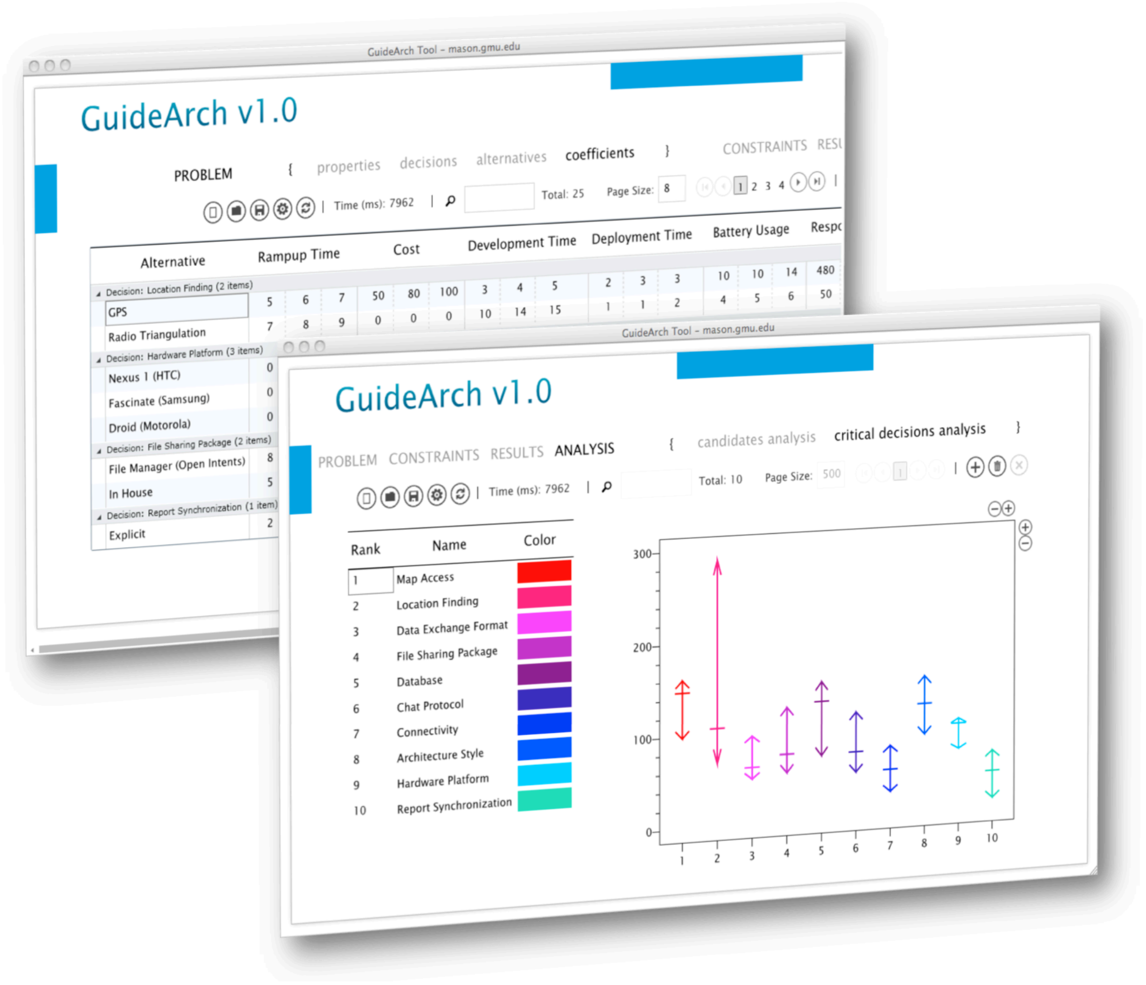
15.4 Reusing POISED

The problem of making early architectural decisions under uncertainty at design-time is very similar to the problem of finding an optimal configuration for the software system under uncertainty at run-time. In both cases, I am dealing with decisions. Moreover, the impact of these decision on quality attributes of the system has uncertainty. The main difference between these two problems is how uncertainty quantified; at run-time, there is an extra step to unify information about all sources of uncertainty under the same theory (i.e., possibility theory). As you may recall, quantifying uncertainty falls outside the scope of this dissertation and I focused on it enough to be able to validate my hypotheses. The only remaining difference is the name of the options selected for decisions at design-time and run-time. In design-time decision making, the options are called alternatives. On the other hand, in run-time decision making, the options are called configurations.

This makes it clear that I can use the very same quantitative framework introduced in this dissertation at design-time. Essentially, I can port POISED, which is targeted at run-time decision making, to design-time decision making. This is exactly what I did in my recent work [36]. The result were very promising and showed success in this transition.

In fact, the problem of design-time decision making is less complex compared to run-time decision making. As I mentioned earlier, at run-time there is an extra step of bringing all assessment under the roof of the same theory. Design-time assessments are already under the same theory (i.e., possibility theory). Moreover, the size of problems usually tend to be larger at run-time. At design-time, decisions are mainly high-level and abstract. On the other hand, at run-time, the decisions are mainly detailed. Therefore, the number of decisions and available options is usually higher. In turn, the search space is larger and takes more time to find the optimal solution. Finally, decisions should be made quickly at run-time. Otherwise, system may underperform for a long period of time. On the other hand, at design-time, there is more time for making decisions. It is desirable to make decisions quickly at design-time. However, it is not a crucial aspect of decision making at design-time.

At design-time, architects are mainly interested to answer what if questions to explore design space. Through this process architects make their assessments more concrete (recall Figure 15.2). This is not the case at run-time, where the goal is to make automatic decisions in a timely manner. Therefore, I extended POISED to help architects see the impact of their decisions. I made the framework more interactive by adding new capabilities. However, these extensions led to a different version of the framework, which I called GuideArch[2]. The detailed description of GuideArch and the extensions are outside the scope of this dissertation. I encourage the interested reader to read my recent work [36] for detailed description of these extensions.


GuideArch Tool – mason.gmu.edu
GuideArch v1.0
PROBLEM
{ properties decisions alternatives coefficients }
CONSTRAINTS
Time (ms): 7962
Total: 25
Page Size: 8
1 2 3 4

Alternative	Rampup Time			Cost			Development Time			Deployment Time			Battery Usage			Resp
Decision: Location Finding (2 items)																
GPS	5	6	7	50	80	100	3	4	5	2	3	3	10	10	14	480
Radio Triangulation	7	8	9	0	0	0	10	14	15	1	1	2	4	5	6	50
Decision: Hardware Platform (3 items)																
Nexus 1 (HTC)	0															
Fascinate (Samsung)	0															
Droid (Motorola)	0															
Decision: File Sharing Package (2 items)																
File Manager (Open Intents)	8															
In House	5															
Decision: Report Synchronization (1 item)																
Explicit	2															

GuideArch Tool – mason.gmu.edu
GuideArch v1.0
PROBLEM CONSTRAINTS RESULTS ANALYSIS
{ candidates analysis critical decisions analysis }
Time (ms): 7962
Total: 10
Page Size: 500
300
200
100
0
1
2
3
4
5
6
7
8
9
10

Rank	Name	Color
1	Map Access	🟥
2	Location Finding	🟥
3	Data Exchange Format	🟪
4	File Sharing Package	🟪
5	Database	🟪
6	Chat Protocol	🟦
7	Connectivity	🟦
8	Architecture Style	🟦
9	Hardware Platform	🟦
10	Report Synchronization	🟩

Figure 15.3: Snapshots of GuideArch in action showing the coefficients (back) and critical decisions (front).

15.5 Implementation

Figure 15.3 shows two snapshots of GuideArch: in the back we have the screen used for getting the architect's input as to the expected range of impact each alternative may have on the system properties, while in the front we have the screen displaying the critical decisions and their impact on the properties of the top ranked architectures. The tool is web-accessible to facilitate sharing, collaboration, and negotiation among the various stakeholders. I used Microsoft® Silverlight™ as my platform, which has appropriate plug-ins for mainstream web browsers. The tool is integrated with Microsoft® Solver Foundation for optimization purposes. The tool also provides several features to enable exploration of the architectural space, including color coding to indicate the importance of decisions and measure the quality of candidate architectures, and ranking of the architectures based on various tunable parameters.

15.6 Evaluation

Evaluation of software architecture research is difficult, as it often hinges on industry participation. I feel fortunate to have had a unique opportunity to employ and evaluate our research in the context of a real world project. GuideArch was used by a team of engineers and academics to explore the architectural space of the SAS project.

Figure 15.4 shows the (flattened) triangular fuzzy value of 10 sample SAS architectures calculated by GuideArch. The horizontal axis marks the architectures' rankings. In SAS, I gave equal weights to the satisfaction of each of all three criteria (i.e., $w_p = w_m = w_o = 1/3$). Looking at Figure 15.4 we can gain insights into the analysis performed by GuideArch. For instance, 1st architecture, which is picked as optimal by GuideArch, has the best combination of three z values, i.e., it has a higher z_m, smaller z_p, and larger z_o than the majority of candidates. As a result, while 1st architecture may be slightly inferior to some candidates with respect to one of the three conditions, it achieves the best set of

[2]Available at http://mason.gmu.edu/~nesfaha2/Projects/GuideArch

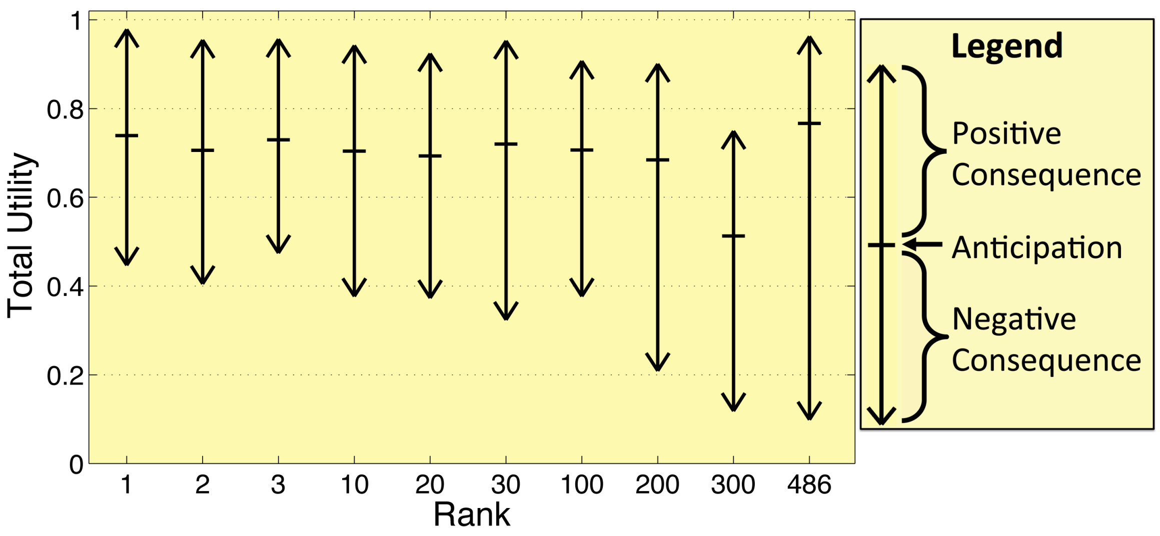


Figure 15.4: The triangular fuzzy value of 10 architectural candidates in SAS.

trade-offs.

To illustrate the benefits of GuideArch, I compare it against what I call the traditional approach. Recall that by traditional approach I collectively refer to any prior research that performs the analysis based on point estimates (i.e., ignores uncertainty). More specifically, here I compare the results of GuideArch against that of ArchDesigner [5], which has aimed to solve a similar problem, albeit without considering uncertainty.

I observed that the traditional approach would have selected the 486th candidate in Figure 15.4 as the optimal solution. Traditional approaches only maximizes the anticipated value (z_m). Comparing the difference between 1st and 486th candidates sheds light on the contributions of GuideArch. 486th architecture achieves the highest z_m among all valid architectures, including the 1st architecture. However, 486th architecture has a very large negative consequence of uncertainty, which has been ignored by the traditional approach. On the other hand, GuideArch selects a solution that has a slightly inferior z_m, but with a better range of uncertainty.

Chapter 16: Conclusion

In this chapter I conclude my dissertation by enumerating the contributions of POISED, threats to validity and avenues for future work.

16.1 Contributions

In this dissertation I provided a detailed description of POISED, my approach for dealing with the problem of making adaptation decisions under uncertainty. I defined related concepts and enumerated the related work. The following is the concrete list of contributions in this research:

- **More effective adaptation decisions:** Since the adaptation decisions are made based on (partial) information about the ranges of uncertainty instead of point estimates, the trade-offs are more accurate and, as a result, the system is more effective in achieving its objectives.

- **Decision making techniques suitable for execution at run-time:** Since the mathematical techniques, which are used in the decision-making process, are efficient and fast, my approach is suitable for execution at run-time.

- **Analysis of uncertainty in self-adaptive software:** This research presents a terminology for referring to uncertainty in self-adaptive software. Moreover, the definition of uncertainty and its sources in self-adaptive software can be used for communication about uncertainty in the community.

- **Tools and algorithms:** I implemented a framework based on the techniques and algorithms resulted from this research. This framework is extensible, as it allows implementation of extensions covering new quality dimensions and systems. It is possible to reuse the framework in new adaptation problems with straightforward customizations. In fact, I have already reused this framework for the problem of decision making at design-time.

- **Validation of research hypotheses:** I used a prototype software system developed on top of Prism-MW to conduct experiments, evaluate the ideas upon which my framework is based, and validate my research hypotheses. The evaluation scripts are publicly available[1] and other researcher can reproduce my evaluation results.

16.2 Threats to Validity

With regard to the internal threats, there is only one issue. The weights of the three objectives (recall Section 10.2.3) impact the outcome of POISED. In fact, in Section 14.6 and Figure 14.6, I showed how selecting different set of weights results in different optimal solutions. One may be tempted to find a golden set of weights that can work best in all domains. However, as I alluded to in Section 14.6, there is no such thing as a set of golden weights. Each system needs its own tuning. For instance, in a mission critical system, where the goal is to guarantee a very narrow range of worst case behavior, it may be desirable to increase w_p. On the other hand, in a system that one good execution out of many tries will be selected, it may be desirable to increase w_o.

An external threat is related to quantification of uncertainty, as the accuracy of POISED heavily depends on the ability to quantifying uncertainty. In Section 9, I presented some techniques and concrete examples to show feasibility of quantification for POISED. In fact, in Section 14.4 and Figure 14.4, I showed how different levels of accuracy in assessing uncertainty impact POISED's ability to find the optimal solution. However, according to hypothesis 3, even partial information about uncertainty allows making better decisions compared to considering only point estimates. Therefore, even in the worst case, POISED

[1]http://www.sdalab.com/projects/poised

would not work worst than traditional approach.

Another external threat is with regard to the analytical models used during my experiments. As I mentioned in Chapter 14, I focused on execution time to be able to produce results that are statistically strong. However, the analytical models used for assessing execution time may not be applicable to other quality attributes. There is an extensive body of literature on analytical modeling (e.g., [69]), and hence, I left analytical modeling outside the scope of this dissertation. POISED is general enough to accommodate any other quality attribute. In fact, the formalism (recall Chapter 8) already models other quality attributes. As I described in Section 13.2.2 and depicted in Figure 13.2, POISED already has the extension points for plugging other analytical models.

16.3 Future Work

I plan to extend this work on two fronts. The first front is considering other sources of uncertainty. One of the important sources of uncertainty is the behavior of the selected solution over time. Figure 16.1a depicts a configuration picked by traditional approach, in which the behavior over time is neglected. On the other hand, Figure 16.1b, depicts my envisioned approach, which considers the behavior of the selected configuration over time, i.e., selects a configuration that has lower utility at the moment in which the decision is made with expectation for improvement in the future. Since the system is expected to do better in the future, this approach decreases the number of adaptations compared to traditional approach. Note that in Figure 16.1, the behavior over time is depicted linearly, but, in general the behavior over time may follow a different trajectory.

The second front for extending this work is application of POISED to other problem domains. I have already done this for the problem of making early architectural decisions. However, there are multiple other problems that can easily be extended to benefit from the contributions of the POISED. One notable example is the problem of component deployment at run-time [64]. In the domain of deployment problem, decisions are allocations of component to physical hosts and the goal is to maximize the utility of the impact of

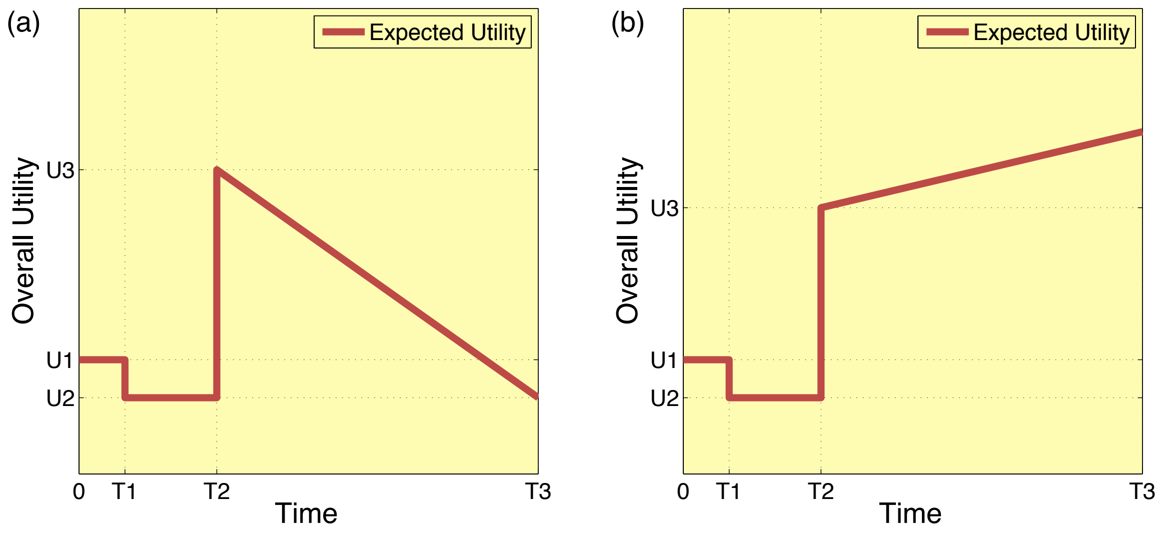


Figure 16.1: The utility of a self-adaptive system based on the decision using: (a) traditional approach, where the behavior over time is not considered and (b) my envisioned approach, which considers the behavior over time.

these decisions on quality attributes. Since the assessment of the impact of redeployment decisions on quality attributes is mired with uncertainty, POISED can naturally and easily be applied to this problem.

Bibliography

Bibliography

[1] Neos server for optimization. http://www-neos.mcs.anl.gov/.

[2] Stochastic programming community home page. http://www.stoprog.org/index.html?spintroduction.html.

[3] Agrawala, A., Krause, J., and Vestal, S. Domain-specific software architectures for intelligent guidance, navigation and control. In *IEEE Symp. on Computer-Aided Control System Design* (Napa, California, Mar. 1992), p. 110116.

[4] Ahmed, S. Introduction to stochastic integer programming, Feb. 2004.

[5] Al-Naeem, T., Gorton, I., Babar, M. A., Rabhi, F., and Benatallah, B. A quality-driven systematic approach for architecting distributed software applications. In *Int'l Conf. on Software Engineering* (St. Louis, Missouri, May 2005), pp. 244–253.

[6] Asadi, M., Esfahani, N., and Ramsin, R. Process patterns for MDA-Based software development. In *Software Engineering Research, management and Applications* (Montreal, Canada, May 2010), pp. 190–197.

[7] Aughenbaugh, J. M. *Managing uncertainty in engineering design using imprecise probabilities and principles of information economics.* PhD thesis, Georgia Institute of Technology, June 2006.

[8] Baresi, L., Pasquale, L., and Spoletini, P. Fuzzy goals for requirements-driven adaptation. In *Int'l Requirements Engineering Conf.* (Sydney, Australia, Sept. 2010), pp. 125–134.

[9] Becker, S., Koziolek, H., and Reussner, R. The palladio component model for model-driven performance prediction. *J. Syst. Softw. 82*, 1 (Jan. 2009), 3–22.

[10] Beraldi, P., and Ruszczyski, A. A branch and bound method for stochastic integer problems under probabilistic constraints. *Optimization Methods and Software 17*, 3 (June 2002), 359382.

[11] Bertsekas, D. P., and Tsitsiklis, J. N. *Introduction to Probability, 2nd Edition.* Athena Scientific, July 2008.

[12] Bruneton, E., Coupaye, T., Leclercq, M., Quma, V., and Stefani, J.-B. The FRACTAL component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *SoftwarePractice & Experience 36*, 11 (Sept. 2006), 12571284. ACM ID: 1152345.

[13] Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F., and Mirandola, R. Qos-driven runtime adaptation of service oriented architectures. In *The joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Amsterdam, The Netherlands, Aug. 2009), pp. 131–140.

[14] Cheng, B., Lemos, R. d., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G. M., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H. M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Muller, H. A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., and Whittle, J. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, LNCS Hot Topics. 2009, pp. 1–26.

[15] Cheng, B. H., Sawyer, P., Bencomo, N., and Whittle, J. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Int'l Conf. on Model Driven Engineering Languages and Systems* (Denver, Colorado, Oct. 2009), pp. 468–483.

[16] Cheng, S. W., and Garlan, D. Handling uncertainty in autonomic systems. In *Int'l Wrkshp. on Living with Uncertainty* (Atlanta, Georgia, Nov. 2007).

[17] Clements, P., Kazman, R., and Klein, M. *Evaluating Software Architectures: Methods and Case Studies.* Addison-Wesley Professional, Nov. 2001.

[18] Coletti, G., and Scozzafava, R. Conditional probability, fuzzy sets, and possibility: a unifying view. *Fuzzy Sets and Systems 144*, 1 (May 2004), 227–249.

[19] Cooray, D., Malek, S., Roshandel, R., and Kilgore, D. RESISTing reliability degradation through proactive reconfiguration. In *Int'l Conf. on Automated Software Engineering* (Antwerp, Belgium, Sept. 2010), pp. 83–92.

[20] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*, 3 ed. MIT Press, July 2009.

[21] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. A generic component model for building systems software. *ACM Trans. Comput. Syst. 26*, 1 (Feb. 2008), 1–42.

[22] De Jong, K. A. *Evolutionary computation: a unified approach.* MIT Press, Cambridge, Mass., 2006.

[23] Doyle, G. S. *A Methodology for Making Early Comparative Architecture Performance Evaluations.* PhD thesis, George Mason University, Dec. 2010.

[24] Dubois, D., and Prade, H. Fuzzy sets and probability : Misunderstandings, bridges and gaps. In *Int'l Conf. Fuzzy Systems* (San Francisco, CA, USA, Apr. 1993), pp. 1059–1068.

[25] Dubois, D., and Prade, H. Possibility theory, probability theory and multiple-valued logics: A clarification. *Annals of Mathematics and Artificial Intelligence 32* (Aug. 2001), 3566. ACM ID: 590454.

[26] Dubois, D., Prade, H., and Sandri, S. On possibility/probability transformations. In *IFSA Conference* (Seoul, Korea, July 1993).

[27] Elkhodary, A., Esfahani, N., and Malek, S. FUSION: a framework for engineering self-tuning self-adaptive software systems. In *Int'l Symp. on the Foundations of Software Engineering* (Santa Fe, New Mexico, Nov. 2010), pp. 7–16.

[28] Elkhodary, A., Malek, S., and Esfahani, N. On the role of features in analyzing the architecture of self-adaptive software systems. In *4th International Workshop on models@runtime* (Denver, Colorado, USA, Oct. 2009).

[29] Epifani, I., Ghezzi, C., Mirandola, R., and Tamburrelli, G. Model evolution by run-time parameter adaptation. In *IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009* (May 2009), IEEE, pp. 111–121.

[30] Esfahani, N. A framework for managing uncertainty in self-adaptive software systems. In *Int'l Conf on Automated Software Engineering* (Lawrence, Kansas, Nov. 2011).

[31] Esfahani, N., Elkhodary, A., and Malek, S. A learning-based framework for engineering feature-oriented self-adaptive software systems. *IEEE Transactions on Software Engineering 39*, 11 (Nov. 2013), 1467–1493.

[32] Esfahani, N., Kouroshfar, E., and Malek, S. Taming uncertainty in self-adaptive software. In *Int'l Symp. on the Foundations of Software Engineering* (Szeged, Hungary, Sept. 2011), pp. 234–244.

[33] Esfahani, N., and Malek, S. On the role of architectural styles in improving the adaptation support of middleware platforms. In *European Conf. on Software Architecture* (Copenhagen, Denmark, Aug. 2010), pp. 433–440.

[34] Esfahani, N., and Malek, S. Social computing networks: a new paradigm for engineering self-adaptive pervasive software systems. In *Int'l Conf on Software Engineering* (Cape Town, South Africa, May 2010), pp. 159–162.

[35] Esfahani, N., and Malek, S. Utilizing architectural styles to enhance the adaptation support of middleware platforms. *Journal of Information and Software Technology 54*, 7 (July 2012), 786–801.

[36] Esfahani, N., Malek, S., and Razavi, K. GuideArch: guiding the exploration of architectural solution space under uncertainty. In *Int'l Conf on Software Engineering* (San Francisco, CA, May 2013).

[37] Esfahani, N., Malek, S., Sousa, J., Gomaa, H., and Menasce, D. A modeling language for activity-oriented composition of service-oriented software systems. In *Model Driven Engineering Languages and Systems* (Denver, Colorado, Oct. 2009), pp. 591–605.

[38] Ewing, J. M., and Menasc, D. A. A meta-controller method for improving run-time self-architecting in SOA systems. In *ACM/SPEC International Conference on Performance Engineering* (Dublin, Ireland, Mar. 2014), pp. 173–184.

[39] Fielding, R. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, Univ. of California Irvine, June 2000.

[40] Fritsch, S., Senart, A., Schmidt, D. C., and Clarke, S. Time-bounded adaptation for automotive system software. In *Int'l Conf on Software Engineering* (Leipzig, Germany, May 2008), ICSE '08, p. 571580. ACM ID: 1368166.

[41] Garlan, D. Software engineering in an uncertain world. In *FSE/SDP Wrkshp. on the Future of Software Engineering Research* (Santa Fe, New Mexico, Nov. 2010), pp. 125–128.

[42] Garlan, D., Cheng, S. W., Huang, A. C., Schmerl, B., and Steenkiste, P. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer 37*, 10 (Oct. 2004), 46–54.

[43] Ghezzi, C., and Tamburrelli, G. Predicting performance properties for open systems with KAMI. In *Int'l Conf on the Quality of Software Architectures* (East Stroudsburg, Pensylvania, June 2009), QoSA '09, Springer-Verlag, p. 7085. ACM ID: 1574269.

[44] Gibbons, J. D., and Chakraborti, S. *Nonparametric Statistical Inference (4th Edition)*, 4th ed. CRC Press, 2003.

[45] Gomaa, H., Hashimoto, K., Kim, M., Malek, S., and Menasc, D. A. Software adaptation patterns for service oriented architectures. In *ACM Symp. on Applied Computing, Dependable and Adaptive Distributed Systems* (Sierre, Switzerland, Mar. 2010), pp. 462–469.

[46] Gomaa, H., and Hussein, M. Software reconfiguration patterns for dynamic evolution of software architectures. In *Working IEEE/IFIP Conf. on Software Architecture* (Oslo, Norway, June 2004), pp. 79–88.

[47] Hayes-Roth, B., Pfleger, K., Lalanda, P., Morignot, P., and Balabanovic, M. A domain-specific software architecture for adaptive intelligent systems. *IEEE Trans. Softw. Eng. 21*, 4 (Apr. 1995), 288–301.

[48] Hoff, P. D. *A First Course in Bayesian Statistical Methods*, 2nd printing. ed. Springer, June 2009.

[49] Huebscher, M. C., and McCann, J. A. A survey of autonomic computing- degrees, models, and applications. *ACM Comput. Surv. 40*, 3 (Aug. 2008), 1–28.

[50] Inuiguchi, M., and Ramik, J. Possibilistic linear programming: a brief review of fuzzy mathematical programming and a comparison with stochastic programming in portfolio selection problem. *Fuzzy Sets Syst. 111*, 1 (Apr. 2000), 3–28.

[51] Kazman, R., Asundi, J., and Klein, M. Quantifying the costs and benefits of architectural decisions. In *Int'l Conf on Software Engineering* (Toronto, Canada, May 2001), pp. 297–306.

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

[53] Kleppe, A., Warmer, J., and Bast, W. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Professional, May 2003.

[54] Koenker, R. *Quantile regression.* Cambridge University Press, 2005.

[55] Kolmogorov, A. *Foundations of Probability.* 1933.

[56] Kramer, J., and Magee, J. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng. 16*, 11 (Nov. 1990), 1293–1306.

[57] Kramer, J., and Magee, J. Self-managed systems: an architectural challenge. In *Int'l Conf. on Software Engineering* (Minneapolis, Minnesota, May 2007), pp. 259–268.

[58] Lai, Y.-J., and Hwang, C.-L. A new approach to some possibilistic linear programming problems. *Fuzzy Sets Syst. 49*, 2 (July 1992), 121–133.

[59] Lemos, R. d., Giese, H., Muller, H. A., Shaw, M., Andersson, J., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cikic, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., Goeschka, K. M., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Litoiu, M., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezze, M., Prehofer, C., Schafer, W., Schlichting, W., Schmerl, B., Smith, D. B., Sousa, J. P., Tamura, G., Tahvildari, L., Villegas, N. M., Vogel, T., Weyns, D., Wong, K., and Wuttke, J. Software engineering for self-adpaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems* (Dagstuhl, Germany, June 2011), R. d. Lemos, H. Giese, H. Muller, and M. Shaw, Eds.

[60] Li, B., Zeng, G., and Lin, Z. A domain specific software architecture style for CSCD system. *SIGSOFT Softw. Eng. Notes 24*, 1 (Jan. 1999), 59–64.

[61] Luke, S., Panait, L., Balan, G., Paus, S., Skolicki, Z., Bassett, J., Hubley, R., and Chircop, A. ECJ: a java-based evolutionary computation research system. *Downloadable versions and documentation can be found at the following url: http://cs. gmu. edu/eclab/projects/ecj* (2014).

[62] Malek, S., Edwards, G., Brun, Y., Tajalli, H., Garcia, J., Krka, I., Medvidovic, N., Mikic-Rakic, M., and Sukhatme, G. S. An architecture-driven software mobility framework. *Journal of Systems and Software 83*, 6 (June 2010), 972–989.

[63] Malek, S., Esfahani, N., Menasce, D. A., Sousa, J. P., and Gomaa, H. Self-architecting software SYstems (SASSY) from QoS-annotated activity models. In *ICSE Workshop on Principles of Engineering Service Oriented Systems* (Vancouver, Canada, May 2009), pp. 62–69.

[64] Malek, S., Medvidovic, N., and Mikic-Rakic, M. An extensible framework for improving a distributed software system's deployment architecture. *IEEE Trans. Softw. Eng. 38*, 1 (Feb. 2012), 73–100.

[65] Malek, S., Mikic-Rakic, M., and Medvidovic, N. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Trans. Softw. Eng. 31*, 3 (Mar. 2005), 256–272.

[66] Meedeniya, I., Moser, I., Aleti, A., and Grunske, L. Architecture-based reliability evaluation under uncertainty. In *Int'l Conf on the Quality of Software Architectures* (Boulder, CO, June 2011), pp. 85–94.

[67] Menasce, D., Gomaa, H., Malek, s., and Sousa, J. SASSY: a framework for self-architecting service-oriented systems. *IEEE Software 28*, 6 (Dec. 2011), 78–85.

[68] Menasce, D. A., Dowdy, L. W., and Almeida, V. A. *Performance by Design: Computer Capacity Planning By Example.* Prentice Hall PTR, Jan. 2004.

[69] Menasce, D. A., Sousa, J. P., Malek, S., and Gomaa, H. QoS architectural patterns for self-architecting software systems. In *Int'l Conf. on Autonomic Computing* (Washington, DC, June 2010), pp. 195–204.

[70] Narayanan, D., and Satyanarayanan, M. Predictive resource management for wearable computing. In *Int'l Conf. on Mobile systems, applications and services* (San Francisco, California, May 2003), p. 113128. ACM ID: 1189041.

[71] Nowak, M., Pautasso, C., and Zimmermann, O. Architectural decision modeling with reuse: challenges and opportunities. In *ICSE Wrkshp on Sharing and Reusing Architectural Knowledge* (Cape Town, South Africa, May 2010), pp. 13–20.

[72] Oreizy, P., Medvidovic, N., and Taylor, R. N. Architecture-based runtime software evolution. In *Int'l Conf. on Software Engineering* (Kyoto, Japan, Apr. 1998), pp. 177–186.

[73] Pautasso, C., Heinis, T., and Alonso, G. JOpera: autonomic service orchestration. *IEEE Data Eng. Bull. 29*, 3 (2006), 32–39.

[74] Perry, D. E., and Wolf, A. L. Foundations for the study of software architecture. *Softw. Eng. Notes 17*, 4 (Oct. 1992), 40–52.

[75] Poladian, V., Garlan, D., Shaw, M., Satyanarayanan, M., Schmerl, B., and Sousa, J. Leveraging resource prediction for anticipatory dynamic configuration. In *Int'l Conf. on Self-Adaptive and Self-Organizing Systems* (Boston, Massachusetts, July 2007), IEEE Computer Society, pp. 214–223.

[76] Poladian, V., Sousa, J. P., Garlan, D., and Shaw, M. Dynamic configuration of resource-aware services. In *Int'l Conf. on Software Engineering* (Scotland, UK, May 2004), pp. 604–613.

[77] Salehie, M., and Tahvildari, L. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst. 4*, 2 (May 2009), 1–42.

[78] Seo, C., Malek, S., and Medvidovic, N. Component-level energy consumption estimation for distributed java-based software systems. In *Int'l Symp. on Component Based Software Engineering* (Karlsruhe, Germany, Oct. 2008), pp. 97–113.

[79] Shaw, M., and Garlan, D. *Software architecture: perspectives on an emerging discipline.* Prentice-Hall, Inc., 1996.

[80] Tajalli, H., Garcia, J., Edwards, G., and Medvidovic, N. PLASMA: a plan-based layered architecture for software model-driven adaptation. In *Int'l Conf on Automated Software Engineering* (2010), p. 467476.

[81] Taylor, R. N., and Hoek, A. v. d. Software design and architecture the once and future focus of software engineering. In *Int'l Conf on Software Engineering* (Minneapolis, Minnesota, May 2007), pp. 226–243.

[82] Taylor, R. N., Medvidovic, N., Anderson, K. M., E. James Whitehead, J., and Robbins, J. E. A component- and message-based architectural style for GUI software. In *Int'l Conf on Software Engineering* (Seattle, Washington, Apr. 1995), pp. 295–304.

[83] Tofan, D., Galster, M., and Avgeriou, P. Capturing tacit architectural knowledge using the repertory grid technique. In *Int'l Conf on Software Engineering* (Waikiki, Honolulu, Hawaii, May 2011), pp. 916–919.

[84] Tracz, W. DSSA (domain-specific software architecture): pedagogical example. *SIGSOFT Softw. Eng. Notes 20*, 3 (July 1995), 49–62.

[85] Trouwborst, A. *Precautionary rights and duties of states.* Martinus Nijhoff, 2006.

[86] Vandewoude, Y., Ebraert, P., Berbers, Y., and D'Hondt, T. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng. 33*, 12 (Dec. 2007), 856–868.

[87] Walsh, W. E., Tesauro, G., Kephart, J. O., and Das, R. Utility functions in autonomic systems. In *Int'l Conf. on Autonomic Computing* (New York, New York, May 2004), pp. 70–77.

[88] Weyns, D., Malek, S., and Andersson, J. FORMS: a formal reference model for self-adaptation. In *Int'l Conf. on Autonomic Computing* (Washington, DC, June 2010), pp. 205–214.

[89] Whittle, J., Sawyer, P., Bencomo, N., Cheng, B. H. C., and Bruel, J.-M. RELAX: incorporating uncertainty into the specification of self-adaptive systems. In *Int'l Requirements Engineering Conf.* (Atlanta, Georgia, Sept. 2009), pp. 79–88.

[90] Winkler, R. L. Uncertainty in probabilistic risk assessment. *Reliability Engineering & System Safety 54*, 2-3 (1996), 127132.

[91] Zadeh, L. A. Fuzzy sets. *Information and control 8*, 3 (June 1965), 338–353.

[92] Zadeh, L. A. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets Syst. 100* (June 1999), 9–34.

[93] Zimmermann, H. J. Fuzzy programming and linear programming with several objective functions. *Fuzzy Sets and Systems 1*, 1 (Jan. 1978), 45–55.

[94] Zimmermann, H.-J. *Fuzzy Set Theory and its Applications (4th Edition)*, 4th ed. Springer, Oct. 2001.

BIOGRAPHY

Naeem Esfahani started his PhD with the Department of Computer Science at George Mason University (GMU) in 2008. His current research mainly focuses on software architecture, autonomic computing, and mobile/distributed software systems. Esfahani received his MS degree in Computer Engineering with an emphasis on Software Engineering from Sharif University of Technology (SUT) in 2008 and his BS degree in Electrical and Computer Engineering with an emphasis on Software Engineering from University of Tehran (UT) in 2005.