Department of Computer Science
George Mason University
Technical Report Series

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/
703-993-1530

Taming Uncertainty in Self-Adaptation through Possibilistic Analysis

Naeem Esfahani nesfaha2@gmu.edu
Ehsan Kouroshfar ekourosh@gmu.edu
Sam Malek smalek@gmu.edu

Technical Report GMU-CS-TR-2010-10

Abstract

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 wide-spread adoption of self-adaption techniques in risk-averse real-world settings. In this paper, we describe an approach, called *POssIbilistic SElf-aDaptation* (*POISED*), for tackling the challenge posed by uncertainty in making adaptation decisions. POISED builds on possibility theory to assess the positive and negative consequences of uncertainty. It makes adaptation decisions that result in the best range of potential behavior. We demonstrate POISED's application to the problem of improving a software system's quality attributes via runtime reconfiguration of its customizable software components. We have extensively evaluated POISED using a prototype of a robotic software system.

Keywords

Uncertainty, Self-Adaptation, Software Architecture

1. Introduction

Self-adaptation has been shown effective in dealing with the changing dynamics of many application domains, such as mobile and pervasive systems. In response to changes, a self-adaptive software system modifies itself to maintain certain objectives [1,12]. While the benefits of such systems are plenty, their development has shown to be significantly more challenging than traditional software systems [1]. One key culprit is that self-adaptation is subject to *uncertainty* [1].

We distinguish between the external and internal uncertainty. *External uncertainty* arises from the environment or domain in which the software is deployed. For example, the external uncertainty for a software system deployed in an unmanned vehicle may include the likelihood of certain weather conditions occurring. Software self-adaptation is one approach in dealing with the effects of external uncertainty, e.g., in a snow storm the vehicle's navigator component may be replaced with a more conservative navigator to avoid a collision. On the other hand, *internal uncertainty* is rooted in the difficulty of determining the impact of adaptation on the system's quality objectives, e.g., determining the impact of replacing a software component on the system's responsiveness, battery usage, etc. While both sources of uncertainty are of great concern, in this paper we focus our attention on the challenges posed by *internal uncertainty*.

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 [3]. Consider that users often find it difficult to accurately express their quality preferences using complex utility functions, 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 runtime, and so on. All of these factors challenge 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.

The research community has made great strides in tackling the complexity of constructing self-adaptive software systems [1,12]. However, as corroborated by others [1], there is a dearth of applicable techniques for dealing with uncertainty in this setting. A few researchers have recently begun to address uncertainty issues in requirements specification [2,21] and resource prediction [17], but no approach we are aware of has tackled the challenge posed by uncertainty in making adaptation decisions. We believe this has been one of the primary obstacles to wide-spread adoption of self-adaptation in risk-averse domains.

This is precisely the challenge we have aimed to address in this paper. We present a general quantitative approach for tackling the complexity of automatically making adaptation decisions under uncertainty, called *POssIbilistic SElf-aDaptation* (*POISED*). Estimates of uncertainty in the elements comprising a self-adaptive software system are incorporated in *possibilistic analysis* of the adaptation choices. Possibilistic analysis is founded on the principles of fuzzy mathematics [22], which provides a sound basis for representing uncertainty, as well as dealing with its negative and positive consequences on the adaptation choices. POISED redefines the conventional definition of optimal solution to one that has the best range of behavior. In turn, the solution selected by POISED 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.

We demonstrate POISED by applying it to the problem of improving a software system's quality attributes via runtime reconfiguration of its customizable software components. We have evaluated POISED under numerous circumstances and using a prototype of a robotic software system. The results demonstrate POISED's ability to deal with uncertainty by making adaptation decisions that are superior to those of the conventional approach.

The remainder of this paper is organized as follows. Section 0 exemplifies the sources of uncertainty via a robotic software system. Section 3 provides an overview of POISED. Section 0 formally presents the self-adaptation problem that we have used to describe and evaluate POISED. Section 5 describes several techniques for quantifying uncertainty in self-adaptive software. Section 6 presents our possibilistic analysis approach. Section 7 details the evaluation of our work. The paper concludes with an overview of the related literature and avenues of future research.

2. Motivationg Example

We use a subset of a robotic software system developed in our previous work [13] to motivate and describe this research. The robotic software is part of a distributed search and rescue system aimed at supporting the government agencies in dealing with emergency crises (e.g., fire, hurricane). Figure 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 1b indicates the *Maneuver* execution scenario, which aims to prevent the robot from hitting obstacles. The *Camera* feed is sent to *Obstacle Detector*, which runs an image processing algorithm to detect obstacles. This information in turn is sent to *Navigator* to plan the change in the direction. Finally, the navigation plan is sent to the *Controller* to be put into effect.

The software components comprising this system are customizable in the sense that they can be configured to operate in different modes of operation. Figure 1a shows some of the available configuration dimensions for the robot's software components. 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 in turn impact the properties of the whole system. For instance, given the resource constrained nature of the mobile robotic system, it is reasonable to expect the configuration decisions of each component to have a significant impact on the system's performance as well as its battery life. Such configuration decisions often cannot be made prior to the system's deployment, since the runtime properties (e.g., available network bandwidth) may not be known ahead of time. One approach to solving this problem is to determine the optimal configuration at runtime and effect it through self-adaptation of the system.

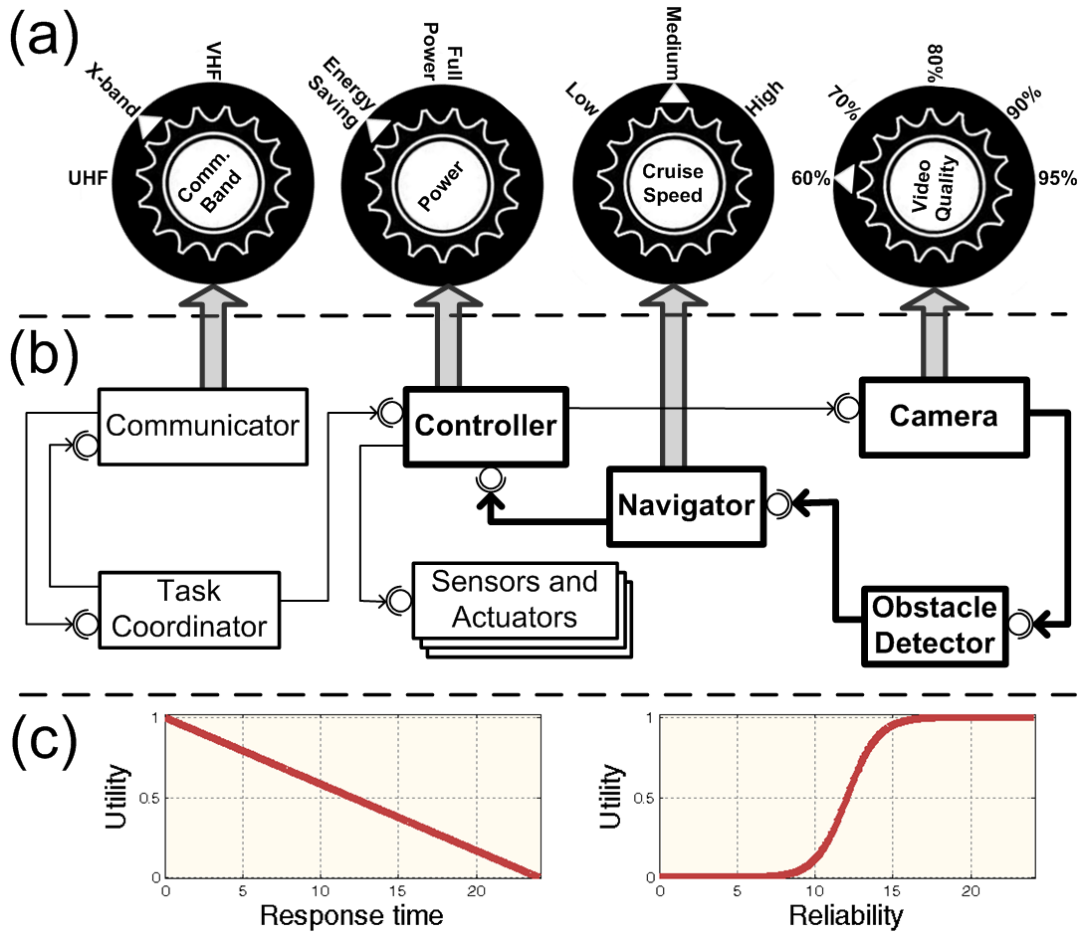

Figure 1. A subset of the robotic software system: (a) configuration dimensions and alternatives for components of the robot, (b) software architecture, where the bold path indicates the components participating in the Maneuver execution scenario, and (c) utility functions defined in terms of quality attributes.

As shown in Figure 1c, for making runtime decisions, utility functions capturing the user's satisfaction with different levels of quality attribute (e.g., availability) are used. The adaptation logic would use analytical models to estimate the effect of each configuration decision on the system's quality attributes and find a configuration that achieves the maximum utility. For example, an analytical model may be used to quantify the response time for *Maneuver* scenario given the configuration of its components.

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 form 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. Both the estimation error and its variation contribute to the uncertainty of adaptation decisions made using such models.

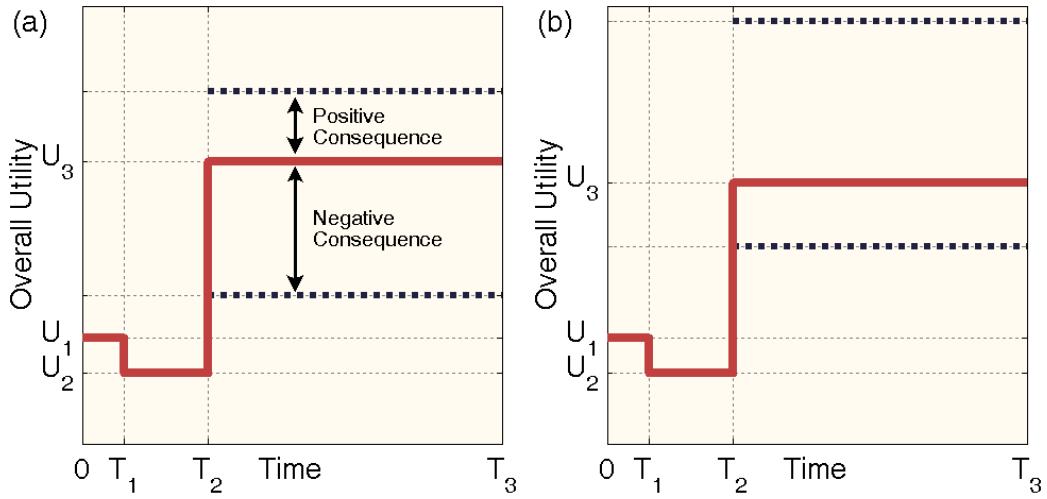

Figure 2. Impact of uncertainty on the utility of a self-adaptive software system: (a) traditional selection of optimal configuration, (b) POISED's selection of optimal configuration.

- *Uncertainty in User Preferences*: Eliciting user's preferences in terms of utility functions, such as those depicted in Figure 1c, is a well-known challenge [1]. 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.

3. Approach

Figure 2a shows the typical behavior of a self-adaptive system that does not incorporate uncertainty in its analysis, which in this paper we abstractly refer to as the *traditional approach*. 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. During 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 2a, this corresponds to U_3, which represents the *expected* utility of the best configuration for the system. In practice, however, the *actual* utility of the system may vary between the two dashed lines, representing the likely positive and negative consequences of uncertainty during T_3. By not accounting for uncertainty in the analysis, the approach is vulnerable to gross overestimation of the utility.

The centerpiece of POISED is the reconceptualization of what is traditionally considered as the optimal solution (adaptation decision), such that the uncertainty is incorporated into the analysis. We illustrate the insights underlying POISED using Figure 2b. Similar to the scenario of Figure 2a, a new configuration is effected at time T_2, except this time we 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.

The details of our approach, including how the likelihood of each objective is calculated, are described in Sections 5 and 6. However, we can make several general observations. As depicted in Figure 2, concurrent satisfaction of the three objectives outlined above may result in a smaller value of *expected* utility (i.e., U_3) in POISED compared to that of the traditional approach. But since the information used to estimate the expected utility is uncertain, *expected* utility is not guaranteed to occur in practice. We argue the true quality of a solution is determined by the range of possible utility. As depicted here and evaluated in Section 7, POISED's objective is to find solutions with a better range.

The above discussion assumes one could quantify the range of utility under uncertainty. There are two general approaches 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 [22]. 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 [22].

For an illustration of this difference, consider Figures 3 and 4 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 5. Possibility distribution models a *fuzzy variable*, while probability distribution models a *random variable*. Possibility distribution in Figure 3 returns the *degree of membership*, while probability distribution in Figure 4 returns the *probability density*.

While uncertainty can be quantified using both approaches, POISED relies on possibility theory in its analysis. Possibility theory is widely used for handling uncertainty in many fields of engineering, including control theory, robotics, and artificial intelligence. One advantage of possibility theory is that the representation of uncertainty and the operations defined on it are more convenient than that of probability theory. For example, performing simple algebraic operations, such as addition and subtraction, on random variables require special considerations (e.g., Central Limit Theorem), and are not always possible. On the contrary, in fuzzy mathematics, which is founded on possibility theory, variables can be simply operated on using traditional algebraic operators. Moreover, in general, fuzzy mathematical problems can be solved much more efficiently than statistical problems [10], making them desirable in many practical engineering problems, including self-adaptive systems. In the remainder of this paper we describe how POISED's research objectives established earlier in this section can be realized through possibilistic analysis.

4. Self-Adaptation Problem

In this section, we provide a formal specification of the self-adaptation problem introduced in Section 0. As mentioned earlier, we use the following adaptation problem as an example for demonstrating and evaluating POISED. However, note that the underlying concepts and techniques in POISED are generally applicable to any self-adaptation problem.

4.1 Configuration

A system like the one depicted in Figure 1b consists of several software components, which we denote as set C. Each component $c \in C$ may have several configuration *dimensions*, which we denote as set D_c. Configuration dimensions correspond to the knobs depicted in Figure 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 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.

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

$$ConfSpace_c \stackrel{\text{def}}{=} \otimes_{d \in D_c} \otimes_{a \in A_d} dom(x_{c,d,a})$$

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*, which we denote as set S. Each scenario $s \in S$ involves a subset of the system's components (i.e., $s \subseteq C$). For instance, the *Maneuver* scenario in Figure 1b consists of the four bolded components. The configuration space of each scenario is the Cartesian product of the configuration space of all the components that participate in it:

$$ConfSpace_s \stackrel{\text{def}}{=} \otimes_{c \in s} ConfSpace_c$$

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

$$ConfSpace \stackrel{\text{def}}{=} \otimes_{c \in C} ConfSpace_c$$

4.2 Quality Attribute

We use Q to denote the set of quality attributes, which are quantifiable non-functional properties (e.g., response time) of interest in the system. A quality attribute may take either discrete or continuous values. For example, 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 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. We represent an analytical formula estimating the configuration's impact on quality attribute $q \in Q$ of execution scenario $s \in S$ as:

$$\widetilde{QE_{s,q}}: ConfSpace_s \longrightarrow dom(q)$$

The tilde is the conventional notation for representing uncertainty. Detailed description of analytical models for estimating quality attributes is beyond the scope of this paper. Numerous previous studies (e.g., [5,7,15]) have developed analytical approaches for estimating quality attributes in terms of the system's architectural configuration. Regardless of the approach, the analytical models are only estimates, and thus a source of uncertainty.

4.3 Resource

We use set R to denote the different computing resources (e.g., memory, CPU) utilized by the software system. For each resource $r \in R$, we 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. We represent an analytical formula estimating the configuration's impact on the system's resource $r \in R$ as follows:

$$\widetilde{RE_r}: ConfSpace \longrightarrow dom(r)$$

Numerous previous studies (e.g., [17-19]) 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.

4.4 User Preference

Similar to the previous research [7,17,18,20], we 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 follows:

$$\widetilde{UP_{s,q}}: ran\left(\widetilde{QE_{s,q}}\right) \longrightarrow [0,1]$$

A higher value indicates more user satisfaction with the system.

Finally, given a vector $\overrightarrow{cnf} \in ConfSpace$, we define the *overall utility* $\widetilde{U}$ to be the cumulative satisfaction of all the user-specified preferences:

$$\widetilde{U} \stackrel{\text{def}}{=} \sum_{\forall s \in S} \sum_{\forall q \in Q} \widetilde{UP_{s,q}}\left(\widetilde{QE_{s,q}}\left(\overrightarrow{cnf_s}\right)\right) \quad \text{Eq. 1}$$

Where $\overrightarrow{cnf_s}$ is defined as the *projection* of $\overrightarrow{cnf}$ from $ConfSpace$ onto $ConfSpace_s$: $\overrightarrow{cnf_s} = proj_{\overrightarrow{I_s}}\left(\overrightarrow{cnf}\right)$, 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 Eq. 1 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.

4.5 Optimization Problem

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

$$argmax_{\left(\overrightarrow{cnf} \in ConfSpace\right)} \widetilde{U} \quad \text{Eq. 2}$$

The solution maximizing the above objective should satisfy two constraints. First, ensure that the solution satisfies the mutual exclusive relationship among the configuration alternatives:

$$\forall c \in C, d \in D_c, \quad \sum_{a \in A_d} x_{c,d,a} = 1 \quad \text{Eq. 3}$$

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

$$\forall r \in R, \quad \widetilde{RE_r}\left(\overrightarrow{cnf}\right) \leq \widetilde{Capacity_r} \quad \text{Eq. 4}$$

For simplicity, the above formulation assumes the capacity of all resources is uncertain. But as you may recall from Section 4.3, this may not always be the case.

5. Quantifying Uncertainty

Before uncertainty can be dealt with in the analysis, we need to be able to quantify uncertainty for a given configuration of the system. POISED's accuracy depends on the ability to: (1) identify the sources of uncertainty, and (2) estimate the level of uncertainty. To put it boldly, our approach addresses "known unknowns", and not "unknown unknowns". However, even if the two conditions are partially satisfied (i.e., only some sources of uncertainty are identified and

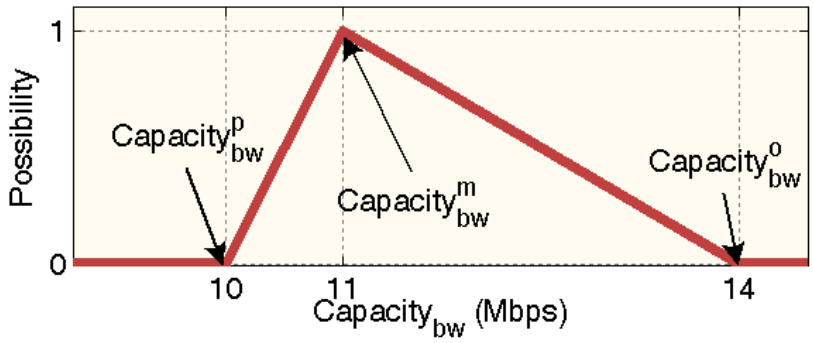

Figure 3. Triangular possibility distribution.

estimated), POISED produces better results than the traditional approach through the incorporation of known uncertainties.

We describe two exemplifying ways of estimating uncertainty in an adaptation problem: eliciting it from the stakeholders (e.g., user, engineer, domain expert), and observing it in the running system. These techniques are not intended to be exhaustive, or even generally applicable, but rather concrete examples to illustrate the feasibility of our work. In this section, we first describe how the uncertainty in both user-specified and monitored elements of the system can be estimated and represented. We then describe how the uncertainty in the individual elements can be combined to quantify the uncertainty for the entire system.

5.1 Eliciting Uncertainty from Stakeholders

Stakeholders may provide the input for many different aspects of a self-adaptation problem. 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 [1]. The specification of such utility functions is highly subjective and inevitably prone to uncertainty. The engineer may also provide the input for certain software properties that cannot be easily monitored. For instance, the maximum memory consumed by a software component is a property that may be available from the component's source code, but not easily obtainable through runtime monitoring. Similarly, the engineer may provide the input for certain systems parameters, such as the available network bandwidth, based on a combination of past experiences, hardware specification, 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 their perceived level of variation in the input. For illustration, Figure 3 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) [22], but for simplicity we 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 $Capacity^{p}_{bw}$, and the *most optimistic*, denoted with $Capacity^{o}_{bw}$. In a triangular distribution, as we reach the boundaries, the possibility of getting the expected value drops to 0. The *most possible* value is $Capacity^{m}_{bw}$, which always has the value of 1.

In the example of Figure 3, the engineer sets the most possible value equal to 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 advertized by the network router manufacturer. By connecting the most pessimistic and optimistic points with the most possible point, we arrive at the triangular possibility distribution representing the uncertainty in the network bandwidth. A similar approach could be used for eliciting and quantifying uncertainty in the other types of input specified by the stakeholders.

5.2 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 runtime, and thus would need to be collected 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 (software and hardware) sensors used for data collection. Such an error is in particular unavoidable

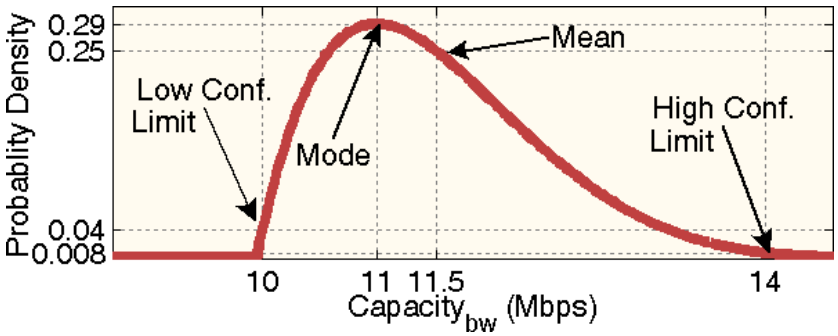

Figure 4. Probability distribution.

with sensors that take samples of a 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. As shown in [7], analytical models used for decision making are abstractions of the system and by definition make simplifying assumptions, which if not held at runtime may make the estimates inaccurate. The inaccuracy of an analytical model could be determined by comparing its estimates (i.e., $\widehat{QE}$, $\widehat{RE}$) against the actual behavior of the system. This can be done either prior to system's deployment by benchmarking the system, or through runtime observation.

We estimate the uncertainty corresponding to any monitored phenomenon as a probability distribution. For example, if the engineer is not able to estimate the uncertainty of network bandwidth using the approach described in Section 5.1, we can estimate it by monitoring the variations and constructing the equivalent probability distribution. Figure 4 shows the probability distribution corresponding to the data collected for variations in the available network bandwidth (i.e., $Capacity_{bw}$). There are numerous approaches for deriving a *probability density function* [9] that represents the probability distribution of collected data, including Q-Q plot [9] that could be used to estimate well-known (e.g., normal, beta) distributions, and Quantile-Regression [11] that could be used to estimate arbitrary complex distributions. In our experiments we found Q-Q plot to be sufficient, as monitoring data often follows one of the well-known distributions.

The majority of existing approaches (e.g., [5,7,15,18]) ignore the probability distribution of the collected data, and simply use the mean value in the analysis. By basing the analysis on the mean behavior of the system, they essentially ignore the statistical characteristics of the data, and thus the underlying uncertainty.

5.3 Quantifying the Overall Uncertainty

From the estimates of uncertainty in the elements comprising a self-adaptive software system, we 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 3, for efficient and effective analysis of uncertainty, we have adopted the possibilistic model of uncertainty in POISED. However, to quantify the overall uncertainty under the possibility theory, and as a fuzzy variable, we also need the uncertainty associated with each of the elements to be expressed as a fuzzy variable.

As you may recall from Section 5.1, 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 3, defines a fuzzy variable. On the other hand, we need to transform the probability distribution representing the uncertainty in monitored elements to the equivalent possibility distribution.

We demonstrate this transformation via the network bandwidth example. From the probability distribution of Figure 4, we can derive the corresponding possibility distribution of Figure 3 as follows: (1) calculate the *confidence interval* [9] and *mode* [9] of probability 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 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 our problem, we quantify the overall uncertainty in system's ability

to satisfy its overall utility (see Eq. 1) via a possibility distribution:

$$\begin{cases} U^p \stackrel{\text{def}}{=} \sum_{\forall s \in S} \sum_{\forall q \in Q} UP^p_{s,q} \left(QE^p_{s,q}(\overrightarrow{cnf_s}) \right) \\ U^m \stackrel{\text{def}}{=} \sum_{\forall s \in S} \sum_{\forall q \in Q} UP^m_{s,q} \left(QE^m_{s,q}(\overrightarrow{cnf_s}) \right) \\ U^o \stackrel{\text{def}}{=} \sum_{\forall s \in S} \sum_{\forall q \in Q} UP^o_{s,q} \left(QE^o_{s,q}(\overrightarrow{cnf_s}) \right) \end{cases}$$

Where $\overrightarrow{cnf} \in ConfSpace$, 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 the sake of simplicity, this formulation assumes the utility functions are monotonic, which is very often the case with the user-specified preferences [20]. If not, by taking the derivative of the functions we could find their extremums; details of which are elided for brevity.

6. Possibilistic Analysis

Now we can describe how the self-adaptation problem of Section 0 can be transformed to a Possibilistic Linear Programming (PLP) [10] problem and solved effectively via conventional mathematical programming solvers.

6.1 Possibilistic Formulation of the Problem

As you may recall from Section 3, 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. This is achieved by pursuing three concurrent objectives: (1) select a configuration that maximizes $z_m \stackrel{\text{def}}{=} U^m$, (2) minimize negative consequence of uncertainty $z_p \stackrel{\text{def}}{=} |U^m - U^p|$, and (3) maximize positive consequence of uncertainty $z_o \stackrel{\text{def}}{=} |U^o - U^m|$. 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. We thus rewrite the objective of Eq. 2 as a PLP problem as follows:

$$\begin{cases} argmin_{(\overrightarrow{cnf} \in ConfSpace)} \, z_p \\ argmax_{(\overrightarrow{cnf} \in ConfSpace)} \, z_m \\ argmax_{(\overrightarrow{cnf} \in ConfSpace)} \, z_o \end{cases} \quad \text{Eq. 5}$$

The next step is the formulation of constraints with uncertainty, which in our problem corresponds to Eq. 4 that ensures the resource usage does not exceed the available capacity. In Section 5.3, we described how the range of uncertainty in resource estimate and capacity can be quantified in the form of a triangular possibility distribution. Since we are dealing with a constraint, we would like to assess it under the worst case scenario. Using the possibility distributions, we can reformulate Eq. 4 as follows:

$$\forall r \in R, \quad RE^p_r(\overrightarrow{cnf}) \leq Capacity^p_r \quad \text{Eq. 6}$$

In the above formulation, RE^p_r is the maximum expected resource usage, while $Capacity^p_r$ is the minimum expected resource capacity. Note that while both RE^p_r and $Capacity^p_r$ 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 4.3, the capacity of certain resources may be a crisp value (e.g., available physical memory), in which case we would simply replace $Capacity^p_r$ with the crisp value of $Capacity_r$ in Eq. 6.

6.2 Solving the Possibilistic Problem

The PLP problem is an instance of a Multi-Objective Linear Programming (MOLP) problem, and to solve it using commonly available linear programming solvers, we first need to transform it to an equivalent Single-Objective Linear Programming (SOLP) problem. To that end, we use Zimmermann's linear membership function [22], which is a linear function μ for each objective z that normalizes its output between 0 and 1:

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

However, for defining each function μ normalizing the output of z, we 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 we are dealing with a minimization objective (i.e., z_p in Eq. 5).

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

$$\begin{cases} z^{PIS}_p \stackrel{\text{def}}{=} \underset{(\overrightarrow{cnf} \in ConfSapce)}{argmin} \, z_p \\ z^{NIS}_p \stackrel{\text{def}}{=} \underset{(\overrightarrow{cnf} \in ConfSapce)}{argmax} \, z_p \\ z^{PIS}_m \stackrel{\text{def}}{=} \underset{(\overrightarrow{cnf} \in ConfSapce)}{argmax} \, z_m \\ z^{NIS}_m \stackrel{\text{def}}{=} \underset{(\overrightarrow{cnf} \in ConfSapce)}{argmin} \, z_m \\ z^{PIS}_o \stackrel{\text{def}}{=} \underset{(\overrightarrow{cnf} \in ConfSapce)}{argmax} \, z_o \\ z^{NIS}_o \stackrel{\text{def}}{=} \underset{(\overrightarrow{cnf} \in ConfSapce)}{argmin} \, z_o \end{cases}$$

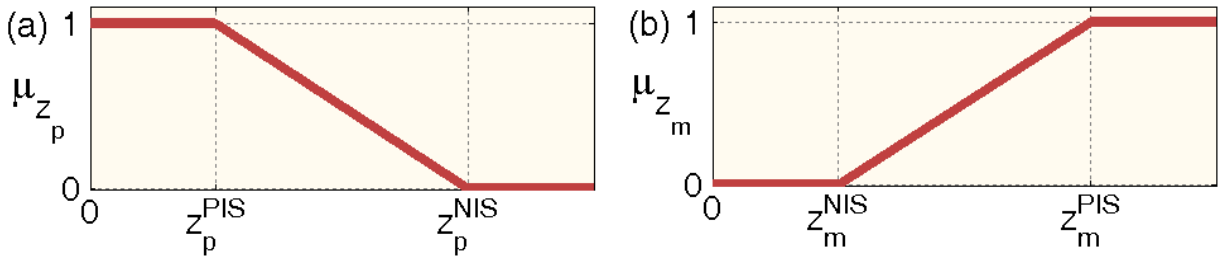

Figure 5. The Zimmermann's linear membership functions: (a) the membership function for z_p, and (b) the membership function for z_m.

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

$$\mu_{z_p}\begin{cases} 1 & z_p < z_p^{PIS} \\ \dfrac{z_p^{NIS} - z_p}{z_p^{NIS} - z_p^{PIS}} & z_p^{PIS} \le z_p \\ & z_p \le z_p^{NIS} \\ 0 & z_p > z_p^{NIS} \end{cases}$$

$$\mu_{z_m}\begin{cases} 1 & z_m > z_m^{PIS} \\ \dfrac{z_m - z_m^{NIS}}{z_m^{PIS} - z_m^{NIS}} & z_m^{NIS} \le z_m \\ & z_m \le z_m^{PIS} \\ 0 & z_m < z_m^{NIS} \end{cases}$$

Function μ_{z_o} is specified similar to μ_{z_m}. Figure 5 shows two instances of μ_{z_p} and μ_{z_m}that normalize the possible outputs of z_p and z_m, respectively.

We can now specify the SOLP problem equivalent to MOLP problem of Eq. 5 as follows:

$$argmax_{(\overrightarrow{cnf} \in ConfSapce)}\ \varphi$$
$$\textit{Subject to:} \quad j \in \{p, m, o\}, \mu_{z_j} \ge \varphi$$

In the above formulation, we have reformulated the objective to maximize the auxiliary 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.

In the above formulation, the three objectives (expressed as constraints) have the same importance. However, in certain applications domains, some of the objectives may have a higher priority. For instance, in a safety 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.

We 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 (Eq. 3 and 6), can be formulated as follows:

$$argmax_{(\overrightarrow{cnf} \in ConfSapce)}\ \varphi$$
$$\textit{Subject to:} \quad j \in \{p, m, o\}, \quad (1 - w_j)\mu_{z_j} \ge \varphi$$
$$\forall c \in C, d \in D_c, \quad \sum_{a \in A_d} x_{c,d,a} = 1$$
$$\forall r \in R, \quad RE_r^p(\overrightarrow{cnf}) \le Capacity_r^p$$

To give a normalized objective μ_{z_j} more priority over others, we make the corresponding constraint (i.e., $\mu_{z_j} \ge \varphi$) more restrictive than others. For that reason, in the above formulation, we 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.

7. Evaluation

We have evaluated POISED on an extended version of the robotic software system described in Section 0 that was developed in our previous work [13]. The robotic software used in our experiments was comprised of 12 software components/connectors, and 50 different configuration alternatives. In POISED we treat 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 Eq. 1) was 5. We used an implementation of the robotic software running on top of Prism-MW [14], which is a middleware platform with extensive support for runtime monitoring and adaptation. In [13] interested reader may find additional details about the robotic software.

For the experiments we setup a controlled environment to allow us create and measure the effect of uncertainty in the system. For that purpose, we used

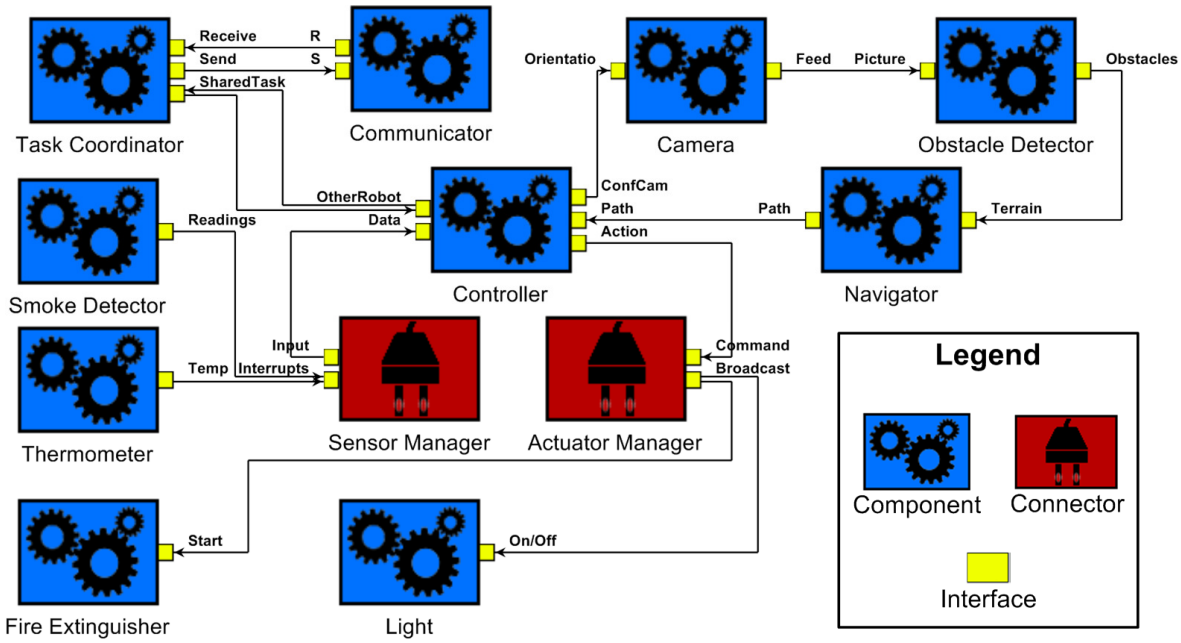

Figure 6. XTEAM model of the robot software.

XTEAM [6], an architectural modeling, analysis, and simulation environment that has been integrated with Prism-MW [14]. Through this integration, the XTEAM models are kept in sync with the software running on top of Prism-MW, and vice versa. Moreover, XTEAM can also be used to control the execution of the software running on Prism-MW, including the ability to fix the workload, and configure the software and hardware properties. We used XTEAM to simulate uncertainty by controlling the extent of random changes in the system parameters (e.g., available network bandwidth, memory consumption of configuration alternatives). However, neither the robotic software nor POISED were controlled, which allowed them to behave as they would in practice. Figure 6 depicts the high-level architectural model of the robot in XTEAM.

The analytical models used in our experiments were derived using the reinforcement learning technique developed in our recent work [7]. The adaptation logic was realized as a three step model interpreter engine that: (1) generates the PLP from the runtime model of the system, (2) solves it using the conventional linear programming solvers (e.g., [16]), and (3) changes the runtime model using the XTEAM's API [6], which automatically effects the changes to the software running on Prism-MW [14].

7.1 Quality Trade-Offs

We compared the quality of solutions selected by POISED with the traditional approach in 10 different experiments. For each experiment, we applied both approaches on the same robotic adaptation problem. As you may recall from Section 3, 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).

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

We show the expected results in Figure 7a. The triangular possibility distribution values correspond to the solution selected by each approach. We observe a similar pattern to what was hypothesized in Figure 2. While POISED's solution may have a slightly lower mode 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.

We complemented the expected ranges of the system's behavior with the actual results obtained in 30 different executions of the system under each

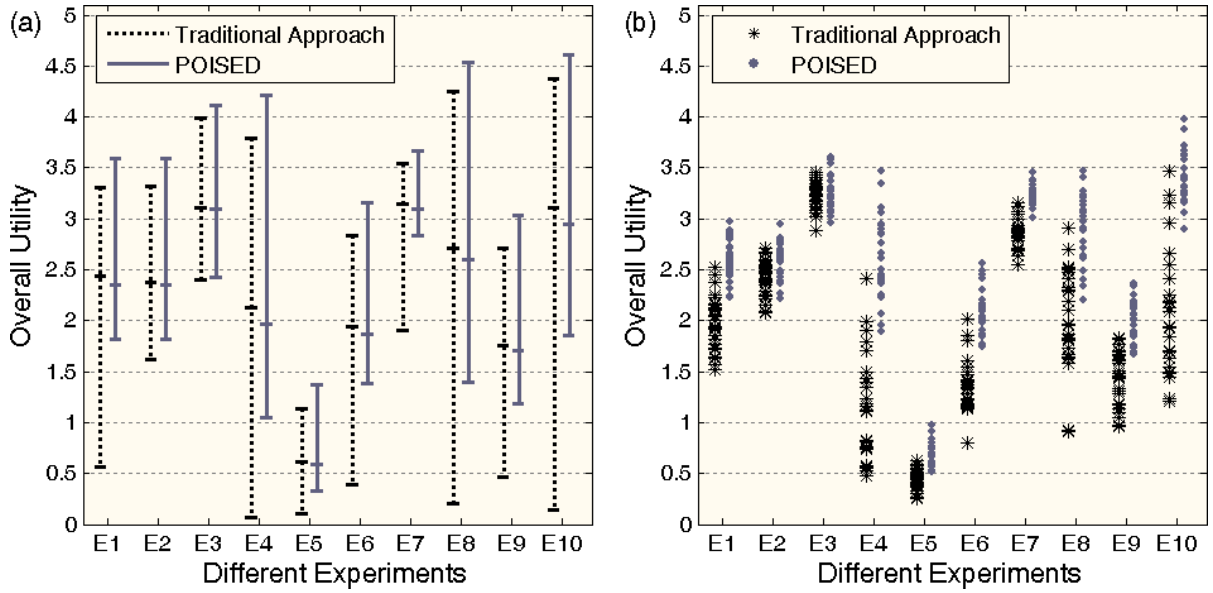

Figure 7. Comparison of POISED with traditional approach in 10 different experiments: (a) possibility distribution for the selected configuration, where a dash marker indicates the most pessimistic, possible, and optimistic values from low to high, (b) 30 actual observations for each selected configuration.

configuration. The results are shown in Figure 7b. For a fair comparison, in each experiment, we used XTEAM to fix the application workload, as well as the range of uncertainty in the execution context (e.g., network bandwidth, memory usage). By "fixing the range of uncertainty" we mean controlling the range of random behavior within each source of uncertainty. Thus, different executions still resulted in different observed behaviors. We can see that the observed utilities are very closely correlated to the corresponding possibility distribution in Figure 7. The results show that in practice POSIED is more likely to select a solution with better overall utility. Note that for a meaningful comparison, in this set of experiments, we did not specify stringent resource constraints, which as shown next could significantly influence the outcome of both approaches.

7.2 Violation of Resource Constraints

We evaluated POISED's ability to satisfy the resource constraints under uncertainty, and compared its results against the traditional approach. We 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 8. The robotic software system corresponding to each selected configuration was then executed 30 times. We instrumented our controlled environment to throttle the available memory. Parenthesized annotations in Figure 8 show the number of times a memory violation was observed in the actual execution of the software system.

We can make several observations from this result. 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 that of traditional approach. On the other hand, since traditional approach ignores the underlying uncertainty in the estimates, as the

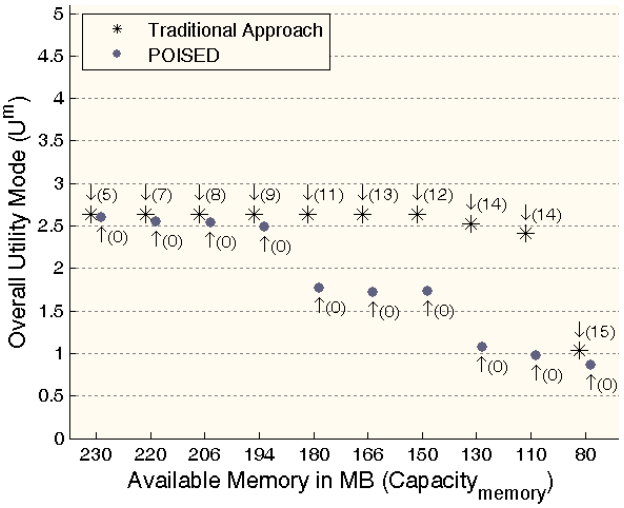

Figure 8. Impact of uncertainty on the overall utility and resource constraints.

available 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 those configurations in $ConfSpace$ that have high utility. Therefore, the traditional approach is forced to select one of the configurations with a relatively low utility. However, even then, since it does not consider the range of uncertainty, 15 of the 30 actual executions of the configuration violate the memory constraint.

7.3 Effect of Weights

In the above experiments, we placed the same amount of weight on each of the three objectives (i.e., $w_p = w_m = w_o = \frac{1}{3}$). However, as you may recall from Section 6.2, this may not always be the case. We evaluated the sensitivity of POISED 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 9 shows the overall utility for the experiments. The solid bar shows the triangular possibility distribution corresponding to the configuration selected by POISED under each weight assignment. The dots depict the observed overall utility as a result of 30 actual execution 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, we see POISED selects a conservative solution, i.e., puts more emphasis on minimizing the negative consequence of uncertainty (recall z_p from Section 6.1). On the contrary, in the two experiments with high w_o, we see POISED selects a risky solution, i.e., puts more emphasis on maximizing the positive consequences of uncertainty (recall z_o from Section 6.1). Both approaches come at the cost of achieving mediocre overall utility mode (most possible).

In the two experiments with high w_m, we see POISED selects a solution with the best overall utility mode (recall z_m from Section 6.1), while ignoring the negative and positive consequences of uncertainty. Finally, 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 considered at the same time, it achieves the best set of trade-offs: very close to the best overall utility mode, and higher possibility of underestimating the overall utility, as opposed to overestimating it. At the same time, we can envision situations in which placing emphasis on one of the objectives may be more appropriate, which POISED allows for naturally.

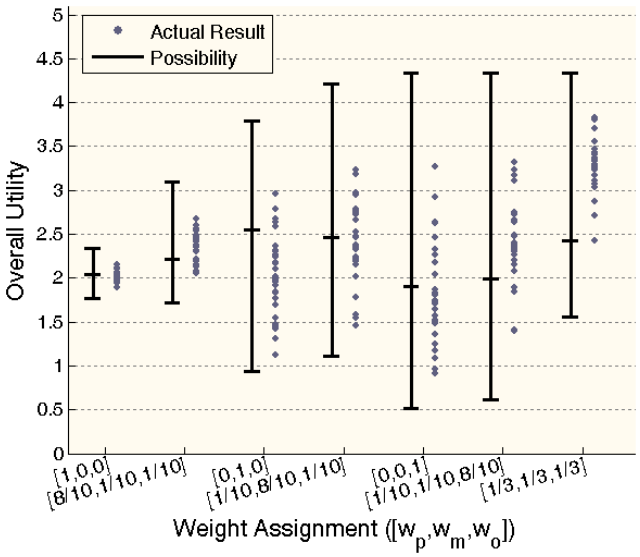

Figure 9. Impact of weights on the selected configuration.

7.4 Sensitivity to Uncertainty Estimates

We performed a set of experiments to evaluate the sensitivity of POISED to the accuracy of uncertainty estimates. Figure 10 shows the results of these experiments. For all of the experiments we used XTEAM to fix the range of uncertainty in the system parameters, as well as the workload. We changed the accuracy of uncertainty estimates used in our analysis.

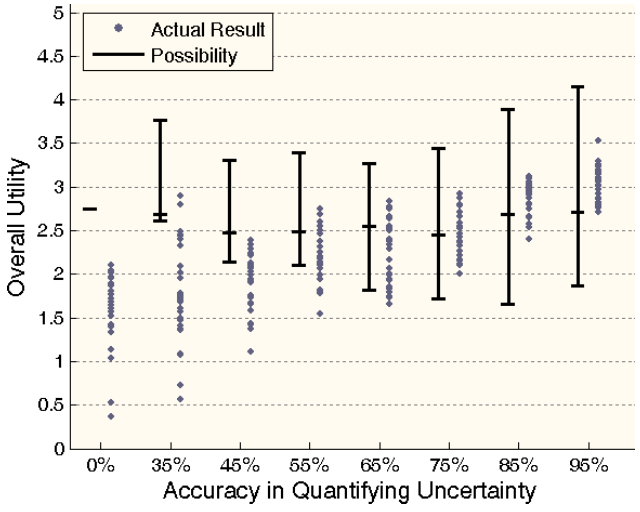

Figure 10. Impact of the accuracy of uncertainty estimates on the quality of POISED solutions.

To that end, we simply changed the confidence level used for transforming the probability distribution corresponding to the monitored data to the equivalent possibility distribution (recall Section 5.3). As one decreases the confidence level in a probability distribution, such as the one depicted in Figure 4, *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 10 denote the accuracy of uncertainty estimates. As we decrease the confidence level from 95% to 0%, thereby making the uncertainty estimates less accurate, POISED makes configuration selections whose overall utilities are not borne out in practice. More specifically, since by decreasing the confidence level we underestimate the uncertainty, the actual results underperform the expected utility. Overestimating uncertainty would have the opposite effect.

Finally, note that in the experiment with 0% accuracy, the most pessimistic, possible, and optimistic points are overlapping. Therefore, by not considering the range of uncertainty, POISED is essentially behaving similar to the traditional approach. Comparing the results of experiment with 0% accuracy to others, corroborates our assertions in Section 5 that even with partially accurate estimates of uncertainty, POISED selects solutions that are better than the traditional approach.

7.5 Performance Trade-Offs

We performed a series of benchmarks to compare the execution time of traditional approach with that of POISED. The results are shown in Table 1. It took POISED longer to compute the optimal solution than that of the traditional approach. This result is not surprising, since as you may recall from Section 6.2, POISED requires 6 additional optimizations to calculate *PIS* and *NIS* values pairs for the three objective functions. While it takes longer to execute POISED, it is still a reasonable approach for our problem. It took 4.6 seconds to find the optimal configuration in a very large problem, consisting of 100 components and 20 different configuration alternatives, for a total of $20^{100} = 1.2 \times 10^{130}$ possible combinations.

Table 1. Execution time of POISED versus traditional approach.

Problem		Execution Time (ms)	
# of Comp	# of Conf	Tradit-ional	POISED
8	**4**	2	30
10	**5**	6	51
18	**7**	10	70
25	**8**	20	180
37	**9**	28	298
50	**10**	30	370
62	**13**	60	630
75	**15**	130	1520
88	**17**	290	3740
100	**20**	400	4600

8. Related Work

The literature in this area of research is extensive. In lieu of enumerating all of the related studies, we refer the reader to [1] and [12] for a comprehensive analysis of the state-of-the-art in self-adaptation. We focus our discussion here to those works that are of utmost relevance. The challenge posed by uncertainty in the construction of dependable self-adaptive software system is an established concept [1]. A few recent works [2-4,7,17,21] have aimed to tackle the different facets of this challenge as follows.

Whittle et al. [21] introduced RELAX, a formal requirements specification language that relies on Fuzzy Branching Temporal Logic to specify the uncertainty inherent in self-adaptive systems. In a subsequent publication [2], Cheng et al. extended RELAX with goal modeling to specify the uncertainty in the objectives. We believe our approach is complementary to their work, as both RELAX and POISED are based on fuzzy mathematics, but target different phases of software life-cycle.

Chuang and Chan [4] presented a QoS management framework that uses a hierarchical fuzzy control model. Their work aims at making it easier for the users to specify their QoS requirements, which are then translated into fuzzy rules. Unlike POISED, their objective is QoS rule satisfaction, and does not target the challenge of making adaptation decisions under uncertainty.

Dynamic configuration of resource-aware services was studied by Poladian et al. [18], where they showed how to select an appropriate set of services to carry out a user task, and allocate resources among those services at runtime. In a subsequent publication, the work was extended to make anticipatory decisions [17], and considered the inaccuracy of future resource usage predictions in making adaptation decisions. Unlike POISED, their approach does not employ possibilistic analysis in incorporating the effect of uncertainty in decisions.

Cheng and Garlan [3] described three specific sources of uncertainty (*problem-state identification*, *strategy selection*, and *strategy outcome*) in self-adaptation and provided high-level guidelines for mitigating them in Rainbow [8]. In this paper, we have presented a novel approach for tackling the challenge of *strategy outcome*, i.e., the impact of uncertainty on the selected solution, and techniques to deal with it.

Finally, in our recent work [7], we presented FUSION, a learning based approach to engineering self-adaptive systems. Instead of relying on static analytical models that are subject to wrong assumptions, FUSION uses machine learning to self-tune the adaptive behavior of the system to unanticipated changes, but does not address making adaptation decisions under uncertainty.

9. Conclusion and Future Work

This paper presented a novel quantitative approach, called POISED, for making adaptation decisions under uncertainty. 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 our work is the reconceptualization of what is typically considered to be the optimal solution as one that has the best range of possible behavior. POISED's analysis can be made as risk-averse as desired via a set of knobs (weights). While POISED is a general approach that can be applied to many types of adaptation problems, it was described and extensively evaluated in the context of a self-adaptation problem aimed at improving a system's quality attributes via runtime reconfiguration of its customizable software components.

Our focus so far has been on the *internal uncertainty*, which is the uncertainty associated with adaptation decisions aimed at satisfying the system's quality objectives. In future, we plan to investigate applicability of POISED to *external uncertainty*, which is the uncertainty associated with decisions aimed at satisfying the domain objectives. We also believe POISED could complement the existing efforts aimed at alleviating uncertainty in other facets of self-adaptation. We envision an integration of RELAX [2,21] with POISED to be a fruitful avenue of future work, as it would allow the traceability of uncertainty from the system's requirements specification to its execution. Proactively adaptive software systems, such as those described in [5,17], face another form of uncertainty—the inaccuracy of future predictions. Investigating the synergy between POISED and such emerging approaches is another interesting avenue of future work.

10. Acknowledgments

This work is partially supported by grant CCF-0820060 from the National Science Foundation.

11. References

[1] Cheng, B. et al. 2009. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems, LNCS Hot Topics*. 1-26.

[2] Cheng, B.H., Sawyer, P., Bencomo, N. and Whittle, J. 2009. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. *Int'l Conf. on Model Driven Engineering Languages and Systems* (Denver, Colorado, October 2009), 468-483.

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

[4] Chuang, S. and Chan, A.T.S. 2008. Dynamic QoS Adaptation for Mobile Middleware. *IEEE Transactions on Software Engineering*. 34, 6 (Dec. 2008), 738-752.

[5] Cooray, D., Malek, S., Roshandel, R. and Kilgore, D. 2010. RESISTing Reliability Degradation through Proactive Reconfiguration. *Int'l Conf. on Automated Software Engineering* (Antwerp, Belgium, September 2010).

[6] Edwards, G., Malek, S. and Medvidovic, N. 2007. Scenario-Driven Dynamic Analysis of Distributed Architectures. *Int'l Conf. on Fundamental Approaches to Software Engineering* (Braga, Portugal, March 2007), 125-139.

[7] Elkhodary, A., Esfahani, N. and Malek, S. 2010. FUSION: A Framework for Engineering Self-Tuning Self-Adaptive Software Systems. *Int'l Symp. on the Foundations of Software Engineering* (Santa Fe, New Mexico, November 2010).

[8] Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B. and Steenkiste, P. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*. 37, 10 (Oct. 2004), 46-54.

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

[10] Inuiguchi, M. and Ramík, J. 2000. 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.

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

[12] Kramer, J. and Magee, J. 2007. Self-Managed Systems: an Architectural Challenge. *Int'l Conf. on Software Engineering* (Minneapolis, Minnesota, May 2007), 259-268.

[13] Malek, S., Edwards, G., Brun, Y., Tajalli, H., Garcia, J., Krka, I., Medvidovic, N., Mikic-Rakic, M. and Sukhatme, G.S. 2010. An architecture-driven software mobility framework. *J. Syst. Softw.* 83, 6 (Jun. 2010), 972-989.

[14] Malek, S., Seo, C., Ravula, S., Petrus, B. and Medvidovic, N. 2007. Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. *Int'l Conf. on Software Engineering* (Minneapolis, Minnesota, May 2007), 591-601.

[15] Menascé, D.A., Sousa, J.P., Malek, S. and Gomaa, H. 2010. QoS Architectural Patterns for Self-Architecting Software Systems. *Int'l Conf. on Autonomic Computing* (Washington, DC, June 2010).

[16] NEOS Server for Optimization. *http://www-neos.mcs.anl.gov/*. Accessed: 08-17-2010.

[17] Poladian, V., Garlan, D., Shaw, M., Satyanarayanan, M., Schmerl, B. and Sousa, J. 2007. Leveraging Resource Prediction for Anticipatory Dynamic Configuration. *Int'l Conf. on Self-Adaptive and Self-Organizing Systems* (Boston, Massachusetts, July 2007), 214-223.

[18] Poladian, V., Sousa, J.P., Garlan, D. and Shaw, M. 2004. Dynamic Configuration of Resource-Aware Services. *Int'l Conf. on Software Engineering* (Scotland, UK, May 2004), 604-613.

[19] Seo, C., Malek, S. and Medvidovic, N. 2008. Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems. *Int'l Symp. on Component Based Software Engineering* (Karlsruhe, Germany, October 2008).

[20] Walsh, W.E., Tesauro, G., Kephart, J.O. and Das, R. 2004. Utility Functions in Autonomic Systems. *Int'l Conf. on Autonomic Computing* (New York, New York, May 2004), 70-77.

[21] Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C. and Bruel, J. 2009. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. *Int'l Requirements Engineering Conf.* (Atlanta, Georgia, September 2009), 79-88.

[22] Zimmermann, H. 2001. *Fuzzy Set Theory and its Applications (4th Edition)*. Springer.