Uncertainty in Self-Adaptive Software Systems

Naeem Esfahani and Sam Malek

Department of Computer Science
George Mason University
{nesfaha2,smalek}@gmu.edu

Abstract. The ever-growing complexity of software systems coupled with their stringent availability requirements are challenging the manual management of software after its deployment. This has motivated the development of self-adaptive software systems. Self-adaptation endows a software system with the ability to satisfy certain objectives by automatically modifying its behavior at runtime. 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. This has been one of the key inhibitors to widespread adoption of self-adaption techniques in risk-averse real-world applications. Uncertainty in this setting is a vaguely understood term. In this paper, we characterize the sources of uncertainty in self-adaptive software system, and demonstrate its impact on the system's ability to satisfy its objectives. We then provide an alternative notion of optimality that explicitly incorporates the uncertainty underlying the knowledge (models) used for decision making. We discuss the state-of-the-art for dealing with uncertainty in this setting, and conclude with a set of challenges, which provide a road map for future research.

Keywords: Self-Adaptive Software Systems, Uncertainty

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 [1–3]. While the benefits of such systems are plenty, their development has shown to be more challenging than traditional software systems [2,3]. One key culprit is that self-adaptation is subject to *uncertainty* [2, 3].

In general, in the field of software engineering, uncertainty is considered as a second-order concept [4]. A common misconception is that by 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 [5], it is typically not possible to eliminate uncertainty altogether as it is not practical nor desirable to collect all of the information about a system. Engineering self-adaptive software is no exception. 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. For instance, one reason behind uncertainty is the fact that the system's user, adaptation logic, and business logic are loosely coupled, introducing numerous sources of uncertainty [6]. 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 runtime, and so on. We refer to these factors as sources of uncertainty. All of these factors challenge the confidence with which the adaptation decisions are made. We believe considering uncertainty as a first-class concept improves the quality or sometimes even the correctness of adaptation decisions.

In spite the fact that uncertainty is prevalent in self-adaptive software systems, it is often considered in an ad hoc fashion. One reason for this is that the term *uncertainty* is a vaguely understood concept in the community, as there are many different sources for uncertainty, and not all sources of uncertainty have similar characteristics.

Some sources of uncertainty are *external*, while others are *internal*. External uncertainty arises from the environment or domain in which the software is deployed. For example, 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 vehicles 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 systems quality objectives, e.g., determining the impact of replacing a software component on the systems responsiveness, battery usage, etc.

Moreover, not all sources of uncertainty have similar characteristics. Sometimes uncertainty is due to lack of knowledge, while other times it is due to the variation in a parameter that affects the adaptation decisions (adaptation parameter). Techniques used to mitigate one type of uncertainty may be different from techniques used to mitigate another type.

In this paper, we aim to change the status quo by first enumerating the common sources of uncertainty in self-adaptive software. We illustrate the sources of uncertainty using a robotic software system developed in our prior work. This also provides the intuition behind the challenges posed by uncertainty in this domain. We provide a more elaborate definition of uncertainty by enumerating its characteristics in the context of prior literature. To that end, we present a conceptual model for better understanding the impact of uncertainty on self-adaptive software. We also present an overview of mathematical techniques commonly used for representing uncertainty and reasoning about it.

The crux of this paper is an intuitive, yet novel, definition of what is considered to be the optimal adaptation decision under uncertainty. Realizing the same definition using fuzzy mathematical techniques in our recent work [7] has produced promising results. Finally, we provide a discussion of the state-of-the-art approaches targeted at addressing the different faces of challenge posed by uncertainty in this setting.

The rest of this paper is organized as follows: Section 2 provides an overview of a self-adaptive robotic application that is used throughout the paper for illustration purposes; Section 3 enumerates the sources of uncertainty in self-adaptive software systems; Section 4 demonstrates the impact of uncertainty on making adaptation decisions; Section 5 establishes a new definition for what is typically considered to be the optimal adaptation decision; Section 6 provides a framework for understanding uncertainty based on its characteristics; Section 7 discusses the commonly used mathematical approaches for representing and reasoning about uncertainty; Section 8 provides an overview of the state-of-the-art in this area; and finally the paper concludes in Section 9 with a summary of contributions and a set of remaining research challenges.

2 Illustrative Example

To demonstrate the ideas and help the discussion, we use a robotic software system that have been developed in our previous work [8] as a running example. The robotic software is part of a distributed search and rescue system [8] aimed at supporting the government agencies in dealing with emergency crises (e.g., fire, hurricane). Fig. 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. Such a system may be comprised of many different execution scenarios. For instance, the bold path in Fig. 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. Fig. 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 runtime, since the system properties

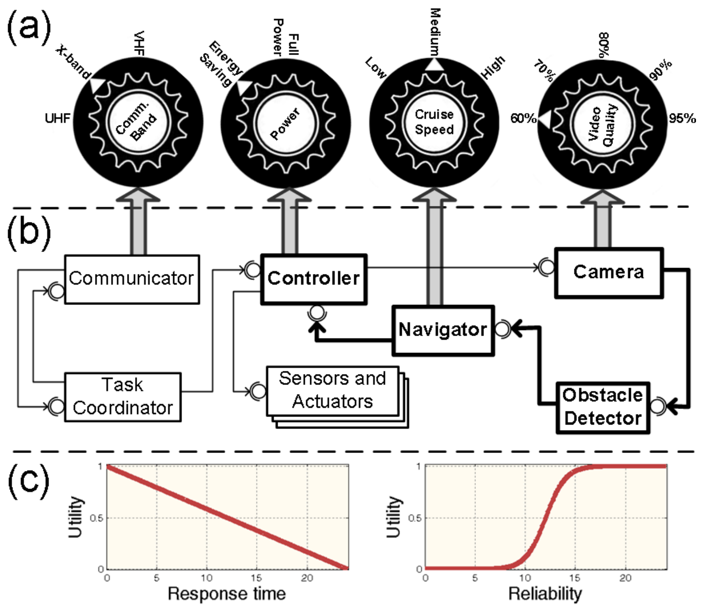

Fig. 1. A subset of the robotic software: (a) configuration dimensions and alternatives for components of the robot, (b) software architecture, and (c) utility functions defined in terms of quality attributes.

(e.g., available bandwidth) are often not known at design-time and may change at runtime.

As shown in Fig. 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 uses analytical models to estimate the effect of configuration decision 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 Queueing Network model [9], may be used to quantify the response time of a particular scenario. The objective of the self-adaptive system is to maintain a configuration for the system that achieves the maximum overall utility.

In the next section, we elaborate on the various forms of uncertainty faced by a self-adaptive software system such as this.

3 Sources of Uncertainty in Self-Adaptive Software

We borrow concepts from FORMS, a reference architecture for self-adaptive software systems developed in our prior work [10], to describe the sources of uncertainty, and exemplify them in the robotics software system. Fig. 2 depicts the high level view of a self-adaptive software system according to FORMS. In this model, the self-adaptive software system can be broken down into two parts: *Meta-Level* and *Base-Level*. The base-level subsystem provides the main functionality of the software (i.e., application logic), while the meta-level subsystem manages the base-level subsystem by reflecting on its behavior (i.e., adaptation logic). Inside the meta-level subsystem we have the MAPE-K feedback control loop [11] from IBM. In this architecture, there are four types of components that operate on the managed subsystem (i.e., base-level) and are devoted to *Monitoring*, *Analysis*, *Planning*, and *Execution* (*MAPE*). MAPE components share various models using what is known as *Knowledge* (*MAPE-K*).

The other two entities in Fig. 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, Fig. 1c shows user's expectations 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 the base-level subsystem interacts with entities from that environment. Since the meta-level subsystem is responsible for keeping the base-level subsystem on track (i.e., ensure it satisfies the user's objectives), it also needs to monitor the environment. For instance, in the robotic software system depicted in Fig. 1, the meta-level subsystem uses sensors to estimate the amount of light in the environment to adjust the *Camera* accordingly.

The entities in Fig. 2 are loosely coupled. The meta-level subsystem needs to use models of other entities in Fig. 2 as their abstractions to make adaptation decisions. The loose coupling between the meta-level subsystem and the other elements of a self-adaptive software (i.e., User, Base-Level, and Environment) is the root cause of uncertainty in self-adaptive software. Sometimes this separation among the elements of a self-adaptive software is unavoidable (e.g., distinction between system and environment), while other times it is simply necessary for enabling reuse and to manage the complexity of constructing such systems (e.g., distinction between managing parts and managed parts of a system [10,12]). We discuss the sources of uncertainty due to this loose coupling as well as a few others in the following:

- **Uncertainty due to simplifying assumptions:** This source of uncertainty is related to the *"Manages"* interface in Fig. 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 that sometimes the assumptions underlying the model are not held at runtime. 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

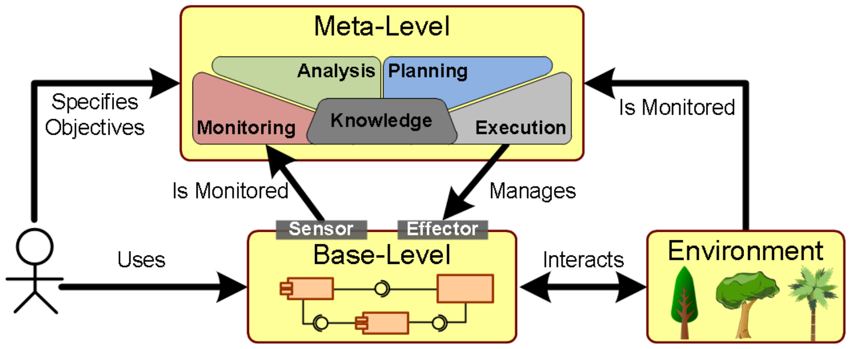

Fig. 2. High level view of self-adaptive software.

also the magnitude of error varies depending on the circumstances. In other words, although the models are not wrong, simplifying assumptions decrease their accuracy.

- **Uncertainty due to model drift:** This source of uncertainty is related to *"Is Monitored"* and *"Manages"* interface. As we discussed earlier, for the sake of generality and reuse, the meta-level subsystem should be separated from the rest of elements in Fig. 2; therefore, due to loose coupling between the meta-level subsystem and base-level subsystem, models (knowledge) used for making decisions in the meta-level subsystem may become inaccurate representations of the base-level subsystem. Another reason for inaccuracy is the adaptation itself. Certain changes may not be enacted exactly as meta-level subsystem requests, creating a drift between the models and actual base-level subsystem. In the above example, consider the scenario in which the meta-level subsystem requests the base-level subsystem to change the communication protocol from TCP to UDP (i.e., replace a connector). If the base-level subsystem fails to enforce this change, the models used for reasoning by the meta-level subsystem become inconsistent representation of the actual base-level subsystem. Compared to the previous source of uncertainty, here we are talking about the models that over time become wrong and do not represent the base-level subsystem correctly.
- **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, even if the actual value of the bandwidth is fixed. This type of uncertainty is referred to as noise to indicate the error in the employed probes.
- **Uncertainty of parameters in future operation:** This source of uncertainty is also related to *"Is Monitored"* interfaces and is due to the actual changes in the monitored phenomenon. Without considering the behavior

of the system in its future operation, a self-adaptive software may not be able to achieve its objective. For instance, our robotic software system uses sensors to measure the amount of light, which may change as the robot navigates a terrain, to adjust the configuration of *Camera* component. The changes in light can be predicted based on the trajectory of robot movement. If the robotic software system does not consider the predictions and make decisions only based on the current amount of light, the adjustments to the *Camera* may not result in optimal improvement. Such a system is also susceptible to continuous adaptation of the system, and loss of stability, as the self-adaptation logic optimizes the system for current operating conditions, which are continuously invalidated due to changes.

- **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 software systems used merely as data processing entities deployed on isolated servers to becoming ubiquitous and engaging the users in their daily activities. These new breeds of software often depend on correct human behavior. However, human behavior is inherently uncertain [4, 13], 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 Fig. 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 expressing users' requirements and eliciting preferences. While the previous source is rooted in software's dependency on human behavior, uncertainty in the objectives is the reverse relationship, i.e., it is related to human's dependency on software. In a large-scale multi-user system, users often have multiple concerns, some of which may be conflicting with one another. Eliciting user's preferences in terms of utility functions, such as those depicted in Fig. 1c, is a well-known challenge [2], as the users often have difficulty expressing their preferences and expectations using mathematical functions. Thus, the overall accuracy of such preferences remains subjective, making the analysis based on them prone to uncertainty.
- **Uncertainty due to decentralization:** In a self-organizing system several meta-level subsystems manage different base-level subsystems [3]. They create a decentralized system, where the knowledge is scattered among the self-organization units comprising the system. A self-organizing unit typically does not have complete control over the actions of other units. 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 decentralized among different entities, which makes the system prone to uncertainty. For instance, in our 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 the area as fast as possible. This high-level collaboration adds to uncertainty as no robot may have complete knowledge of the entire system in real-time and may not be able to control the other robots.

- **Uncertainty in the context:** Many self-adaptive software systems are intended to be used in different execution *contexts*. To that end, the meta-level subsystem is expected 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 of systems in this category. 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 [4], which is subject to change as the context of execution changes. For instance, in the robotic software system, a robot may move to a place in which a barrier shields its signal and prevents it from communicating with other robots, making the status of that robot unknown to the rest of system.
- **Uncertainty in cyber-physical systems:** As computation continues to become cheaper and more widespread, software and physical spaces become increasingly intertwined and tightly integrated. As a result, physical concepts are becoming increasingly important in software systems. In fact, self-adaptation capabilities are often sought after to manage the interactions between software and physical entities. This increases non-determinism and uncertainty in the software due to the fact that the physical world itself is inherently uncertain. Uncertainty caused by the effect of physical world on the software is a subset of context, which was described in the previous source. However, software can also effect the physical world, and this interaction can also host uncertainty. For instance, a robotic software system's ability to maneuver a terrain is not only a function of the accuracy of its software (e.g., routing algorithms), but also the precision in the physical steering components, as well as the physical conditions of the terrain. A self-adaptive software aimed at ensuring the robot's ability to maneuver the terrains would have to take into account the uncertainty due to the interaction between software, hardware, and physical entities in its analysis.

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., model drift) are relatively new and specific to self-adaptive systems, hence new approaches may need to be devised for addressing them.

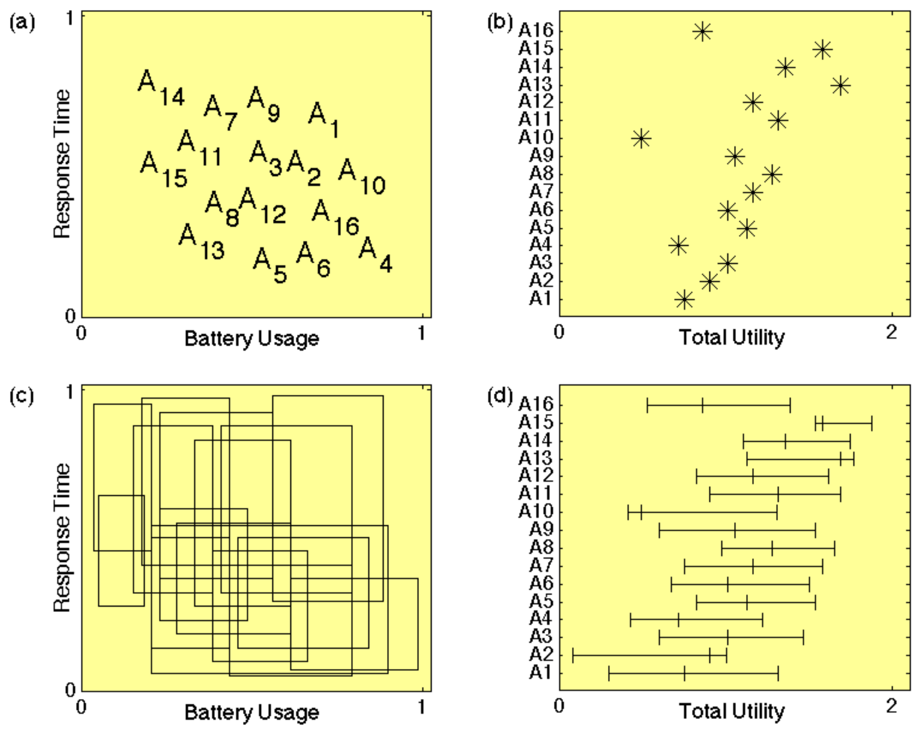

Fig. 3. Impact of uncertainty on the process of making adaptation decisions to satisfy the system's objectives: (a) 16 candidate configurations in a battery usage and response time trade-off, (b) application of utility function to resolve the trade-offs, (c) battery usage versus response time under uncertainty, where each rectangle represents the space of values that an architecture may take, and (d) the range of utility values expected for the 16 configurations under uncertainty.

4 Impact of Uncertainty on Self-Adaptive Software

Uncertainty has a significant impact on a self-adaptive software system's ability to satisfy its objectives. Prior research for the most part have ignored the challenges posed by uncertainty, which hamper their adoption in real-world risk-averse domains. We collectively refer to these as the *traditional approaches*. We illustrate their shortcoming using an instance of the robotic software system in which the objective is to choose from a pool of 16 candidate configurations, such that *battery usage* and *response time* are minimized.

The traditional approaches assume that the impact of candidate configurations on properties of interest can be precisely estimated. If that was the case, then one could visualize the situation as in Fig. 3a. Here, for the sake of clarity, the values for *response time* and *battery usage* are normalized between zero and one. Assuming both properties have the same level of importance, to compare the 16 configurations, for each configuration we first sum up the values obtained from the corresponding utility function. Recall that utility functions are used to quantify the users' preferences with the values attained in properties. Fig. 3b achieves just that, as it shows the overall value for the candidate configurations. In this space, configurations can be compared with one another. For example, we can see that A_{13} is the best configuration, as it obtains the largest total value.

While the aforementioned approach is theoretically sound, it is not useful in practice, as it does not incorporate the underlying uncertainty in every facet of the approach, including the fact that analytical models often cannot precisely quantify the impact of alternative configuration on properties of interest (i.e., there is always some amount of noise), the utility functions may not be accurately representing the users' preferences, etc.

The complexity of incorporating uncertainty in the analysis is shown in Fig. 3c. Here, the uncertainty is represented in terms of range of impact that a configuration candidate may have on the properties of interest. For example, the impact of a given configuration on battery usage is no longer a single number, but rather a range of values. As a result, each configuration candidate may obtain a value anywhere within the area occupied by the corresponding rectangle. Clearly, comparing two configurations with overlapping rectangles is difficult.

The rectangles in Fig. 3c can be transformed to a space where the trade-off analysis can be performed by applying the utility function on the most optimistic and pessimistic behavior of a given configuration. Fig. 3d shows the resulting range of behavior that one would expect, assuming that uncertainty in various facets of the system can be quantified. Unlike the earlier example, it is not clear what is the optimal configuration, as the behavior of each configuration is now specified as a range, and the ranges offer trade-offs. As described in the next section, there is a need for an alternative definition of optimality in this setting that explicitly takes the uncertainty into consideration.

To gain a better appreciation for the complexity of this problem consider that the simple example used in Fig. 3 consists of only 16 configuration candidates and 2 properties of interest, but a typical self-adaptive software system often consists of many more candidates and properties. Manually exploring and solving this problem is a big burden. Incorporating uncertainty into the analysis makes a problem that is already challenging, so overwhelmingly complex that a manual assessment without the appropriate tools and techniques becomes impossible, which has been the motivation for this research.

5 Reconceptualizing Optimality under Uncertainty

We argue that to tackle the complexity introduced by uncertainty, we need to reconceptualize the definition of the optimality in self-adaptation decision making process to account for the uncertainty underlying the analysis. We provide an intuitive overview of a new definition of optimality and use the robotic software example from the previous section to illustrate it.

Figure 4a shows the shortcomings of the prevalent definition of optimality in making adaptation decisions while ignoring uncertainty. The system is initially

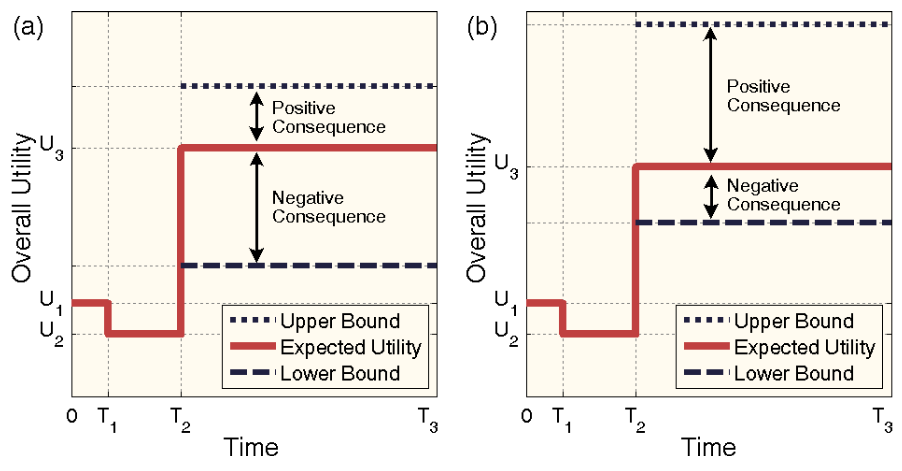

Fig. 4. The utility of a self-adaptive system based on the decision using: (a) traditional definition of optimality, where the uncertainty is not considered, and (b) advocated approach, which considers uncertainty.

executing with utility U_1 prior to time T_1. At time T_1, due to either an internal or external change, the systems 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 Fig. 4a, 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. By not accounting for uncertainty, the approach is vulnerable to gross overestimation of the utility. In other words, the selected optimal solution is rather risky, and in the worst case may be a very poor choice.

We propose an alternative definition of optimality in making adaptation decisions that incorporates uncertainty. Similar to the scenario of Figure 4a, a new configuration is effected at time T_2, except we say a configuration is optimal if it 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 new concept of optimality defined above can be realized using several alternative mathematical approaches (e.g., both probabilistic and fuzzy numbers could be used to indicate the extent of uncertainty). Regardless of how the optimality criteria is realized, we can make a general observation. As depicted in Figure 4, concurrent satisfaction of the three objectives may result in a smaller value of *expected* utility (i.e., U_3) using this approach 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. Therefore, it is reasonable to argue that the true quality of a solution is determined by the range of possible utility.

Furthermore, we argue that the new notion of optimality could be extended to also account for uncertainty in the future operation of a software system. Figure 5a depicts a configuration picked by the traditional approach in which uncertainty in future operation of the system is neglected. As a result, a solution with the highest utility may actually be a very bad choice, since due to uncertainty in future operation of the system, it may in effect obtain a very low utility. Note that for illustration in Figure 5 the behavior over time is depicted linearly, but in general the behavior over time may follow a different trajectory.

Given the variability in system and environmental parameters, an optimal solution is not the one that achieves the highest utility at the point in time in which the decision is made, but the one that anticipates the future behavior (potentially in the form of a probabilistic prediction such as the ones obtained from Hidden Markov Models [14]) of the selected configuration over time. As depicted in Figure 5b, the optimal solution is the one that considers the behavior of the selected configuration over time, i.e., selects a configuration that may have a lower utility at the moment in which the decision is made with the expectation of achieving a better utility over a period of time in future. Another benefit of the new optimality criteria advocated here, but not depicted in the figure, is that since under the reconceptualized notion of optimality the system is expected to maintain a higher utility in its future operation, our approach decreases the number of adaptations compared to the traditional approach. This in essence results in more stable self-adaptive software systems.

These two extension (i.e., Figures 4 and 5) can also be combined. As a result, the range will be formed around the trend line and the size of the range can vary for different points in time.

We believe this new model of reasoning about optimality provides a good foundation for studying the role of uncertainty in self-adaptive software. In our recent work [7] we have used fuzzy mathematical techniques to realize the new model of optimality, which has produced promising results. Our experience shows that the revised definition of optimality increases the accuracy of adaptation decisions, and allows for construction of self-adaptive software that is resilient to fluctuations in the system properties and environmental parameters. While our experience with realizing this approach using fuzzy mathematics has been promising, we believe there are other methods of realizing the approach outlined above (e.g., Bayesian probabilities), as further detailed in Section 7. Finally, as we describe in Section 8, some researchers have already observed the limitation of the existing definition of optimality (i.e., traditional approach) and have investigated possible solutions to this limitation.

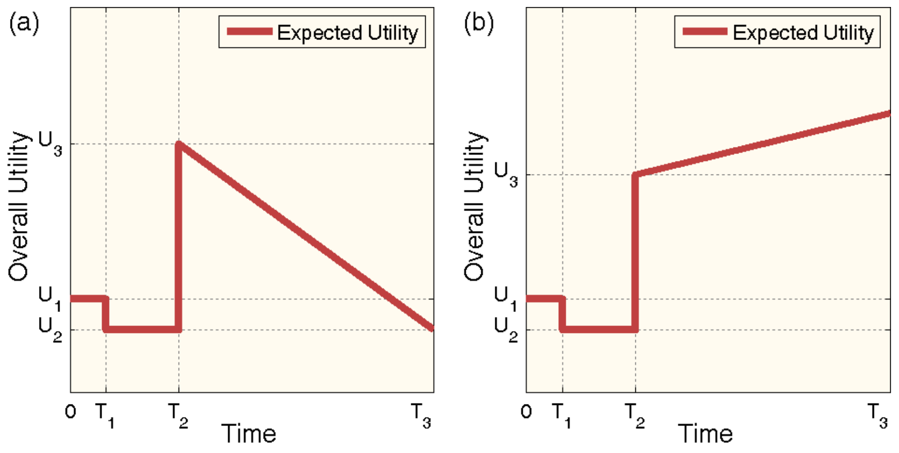

Fig. 5. The utility of a self-adaptive system over time: (a) traditional approach, where the behavior over time is not considered in the selection of a configuration, and (b) advocated approach, which considers the behavior a system in a given configuration over time.

6 Uncertainty Distilled

All sources of uncertainty in self-adaptive software do not have the same characteristics. Although there are some philosophical debates about the true distinction between the different types of uncertainty (e.g., [15]), it is commonly agreed that it is useful to categorize different types of uncertainty in practice. This is because the approaches for modeling different kinds of uncertainty are very different from one another. For instance, often times it is not possible to represent the user's uncertainty in the specification of objectives in terms of utility functions as a probability distribution, since the uncertainty is due to the lack of knowledge, and not variability. In the following subsections we enumerate the different characteristics of uncertainty, which we believe in turn sheds light on the appropriate techniques that should be used to tackle the different sources of uncertainty.

6.1 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 a 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. In this paper, we stay away from philosophical debates as we want to study the practical aspects of uncertainty.

6.2 Variability versus Lack of Knowledge

From a different perspective uncertainty can be categorized as aleatory or epistemic [5]. 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.* [5] 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.

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 in the world is due to limitation of scientific models and hence lack of knowledge [15]. While these arguments are true, we should 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 characteristic of uncertainty.

Both the reducible and irreducible uncertainties can have aleatory and epistemic components. Aleatory and epistemic represent the essence of uncertainty, while irreducible and reducible represent the managerial aspect of uncertainty.

6.3 Spectrum of Uncertainty

Fig. 6 depicts the spectrum 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.*

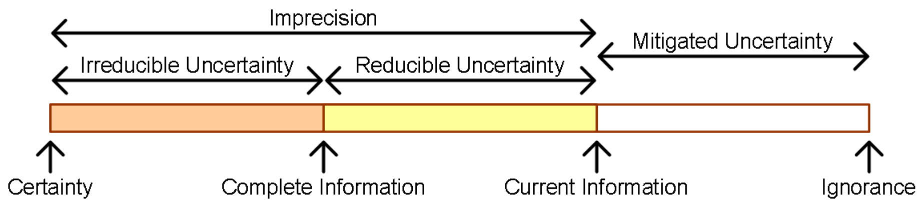

Fig. 6. The spectrum of uncertainty based on the knowledge (adopted and extended from [5]).

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 is 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. We revisit this issue in the next section.

6.4 Characterizing the Sources of Uncertainty

Table 1 characterizes the sources of uncertainty based in relation to the spectrum of uncertainty. To that end, we 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* are due to the lack of knowledge. Be it for the complexity of the models, loose coupling, ambiguity, or distribution, the lack of complete knowledge in these facets of self-adaptation 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

Table 1. Characteristics of different sources of uncertainty.

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

that the behavior of the system may change after the adaptation decision is made.

We drew the conclusions presented in Table 1 from examples of the sources of uncertainty that we have found in the literature, as well as our own prior experiences with the construction of such systems. Some of these examples were enumerated in Section 3. Since it is possible to have several sources of uncertainty in a single phenomenon, uncertainty related to that phenomenon may be both due to variability and lack of knowledge. For instance, one may make a *Simplifying assumptions* and approximate the *Noise* of a given parameter by a well-known probability distribution even if the value of that parameter does not exactly follow the distribution.

7 Mathematical Techniques for Representing and Incorporating Uncertainty

This section provides an overview of two widely applicable approaches for representing and incorporating uncertainty in self-adaptation. As will be described in the next section, existing state-of-the-art has often relied on one of these approaches.

7.1 Probability Theory

Probability theory [16] 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 [17], 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 quite few are aware of philosophical debates regarding different interpretations of probability. Therefore, here we 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 we 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 [18] 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.

7.2 Fuzzy Sets and Possibility Theory

Fuzzy set theory [19] 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 [20] 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 [21, 22] 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.

8 State-of-the-Art

The research community has made great strides in tackling the complexity of constructing self-adaptive software systems [1–3]. However, as corroborated by others [2, 3], there is a dearth of applicable techniques for handling uncertainty in this setting. A few researchers have recently begun to address uncertainty. Table 2 summarizes their work with regard to the sources of uncertainty they are dealing with. In the following subsections we provide an overview of these approaches.

8.1 Rainbow

Cheng and Garlan [6] 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 [12]. 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, a 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 in strategies. Therefore, when Rainbow attempts to select a strategy at runtime, 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 monitor the implementation of the strategy before committing to the change. This is another attribute of the approach that can be modeled using the Stitch language.

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.

8.2 RELAX

Whittle et al. introduced RELAX [23], a formal requirements specification language that relies on Fuzzy Branching Temporal Logic to specify the uncertain requirements in self-adaptive systems (i.e., as indicated in Table 2, RELAX uses possibility theory to deal with the uncertainty of the *Objectives*). 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 runtime. Moreover, the operators capture the kind of uncertainty (*uncertainty factor*) that can initiate the relaxation of requirements.

In a subsequent publication [24], 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.

8.3 FLAGS

FLAGS [25] also uses possibility theory to mitigate the uncertainty of the *Objectives*. Similar to RELAX, FLAGS aims to achieve the basic goal of adaptive systems at the requirements 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), and fuzzy goals using fuzzy temporal language, which in the end is unified with the LTL specification. Therefore, all the software requirements can be specified in a single coherent language.

Table 2. The mathematical theories that are used by existing approaches for dealing with sources of uncertainty.

	Simplifying assumptions	**Model drift**	**Noise**	**Parameters over time**	**Human in the loop**	**Objectives**	**Decentralization**	**Context**	**Cyber-physical systems**
Rainbow	Prob.		Prob.						
RELAX						Poss.			
FLAGS						Poss.			
FUSION	Prob.	Prob.						Prob.	
ADC				Prob.					
RESIST	Prob.	Prob.	Prob.	Prob.				Prob.	
POISED	Poss.		Prob.			Poss.			

8.4 FUSION

FUSION [26] 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. The adaptation actions correspond to variation points in the software that could be exercised at runtime.

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.

8.5 Anticipatory Dynamic Configuration (ADC)

Poladian et al. [27] 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 runtime. The original work did not consider the uncertainty in the environment. Subsequently, the work was extended to make anticipatory decisions [28], and considered the inaccuracy of future resource usage predictions. To that end, they built on the previous work of one of the authors [29] 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. As indicated in Table 2, they use probability theory to achieve this (i.e., Mitigate the uncertainty related to *Parameters over time*).

By considering the resource availability prediction, the anticipatory model of configuration chooses a configuration that maximizes the cumulative expected value of utility over time. This reduces the number of possible future reconfigurations and as a result disruptions 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. On the other hand, 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.

8.6 RESIST

RESIST [14] 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 runtime. 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 through 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 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.

8.7 POISED

POISED [7] is a quantitative approach for tackling the complexity of automatically making adaptation decisions under uncertainty. It builds on possibility theory and fuzzy mathematics to assess both the positive and negative consequences of uncertainty. The goal in POISED is to improve the quality attributes of a software system through reconfiguration of its components to achieve a global optimal configuration for the software system. POISED redefines the conventional definition of optimal adaptation decision 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. This is different from conventional approaches, which do not incorporate uncertainty in their analysis. Such approaches consider the behavior of the system as a point estimate, while POISED consider a range of behavior.

POISED provides a framework to gather and build up uncertainties into a coherent representation, which lends itself well to decision making. POISED relies on Possibilistic Linear Programming to make the trade-off between different configuration alternatives. The configuration knobs in POISED allow the decision maker to specify what aspect of uncertainty is more important: in some cases a solution capable of providing certain guarantees in the worst case scenario would be desirable, in others a solution with higher risk, but the potential of higher quality may be desirable.

9 Conclusion

Uncertainty is a well-known challenge in the construction of dependable self-adaptive software, yet it is a relatively unexplored topic in this area of research. We believe lack of a coherent understanding of uncertainty has hindered the development of suitable techniques to mitigate it. This in turn has prevented the application of solutions developed and evaluated in the academic settings to real-world software systems that are often risk-averse. We believe for widespread adoption of self-adaptation capabilities in real-world application, the research community needs to first develop suitable and practical mechanisms to control the risk associated with self-adaptation of software under uncertainty.

This paper has aimed to address this issue by shedding light on the role of uncertainty in self-adaptive software and distilling its characteristics. We used a robotic software system to illustrate the impact of uncertainty in process of making adaptation decisions, and proposed an alternative method of reasoning about optimality of adaptation decisions that takes imprecision and variability of the knowledge into account. We also provided an overview of the state-of-the-art approaches that have tackled the different facets of uncertainty in self-adaptive software.

While a series of recent publications in this area of research have provided a good foundation for addressing uncertainty issues in self-adaptation, several research challenges remain. One of the most critical issues is that the majority of mathematical techniques for dealing with uncertainty are computationally very expensive. For instance, the standard operations research technique for making decisions under probability theory is called *stochastic programming*. However, stochastic programming is known to be computationally expensive for execution, which makes it unsuitable for use at runtime, where often decisions have to be made very fast.

Another challenge is the ability to quantify uncertainty, which is necessary to be able to reason about uncertainty and adopt the new definition of optimality advocated in this paper (recall Section 5). This is particularly difficult when the uncertainty is in sources that are not necessarily under the control of self-adaptive software (e.g., uncertainty is in the environment). While generally this is a challenging problem that requires further research, our recent work [7] shows that even if uncertainty can only be partially quantified (i.e., roughly estimated), by incorporating it in the analysis, self-adaptation logic is able to make better choices than if it was to completely ignore uncertainty.

Acknowledgments. This work is partially supported by grant CCF-0820060 and CCF-1217503 from the National Science Foundation and grant N11AP20025 from Defense Advanced Research Projects Agency.

References

1. Kramer, J., Magee, J.: Self-Managed systems: an architectural challenge. In: Int'l Conf. on Software Engineering, Minneapolis, Minnesota (2007) 259–268
2. 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., Whittle, J.: Software engineering for Self-Adaptive systems: A research roadmap. In: Software Engineering for Self-Adaptive Systems, LNCS Hot Topics. (2009) 1–26
3. 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., Wuttke, J.: Software engineering for Self-Adpaptive systems: A second research roadmap. In Lemos, R.d., Giese, H., Muller, H., Shaw, M., eds.: Software Engineering for Self-Adaptive Systems, Dagstuhl, Germany (2011)

4. Garlan, D.: Software engineering in an uncertain world. In: FSE/SDP Wrkshp. on the Future of Software Engineering Research, Santa Fe, New Mexico (2010)
5. Aughenbaugh, J.M.: Managing uncertainty in engineering design using imprecise probabilities and principles of information economics. PhD thesis, Georgia Institute of Technology (2006)
6. Cheng, S.W., Garlan, D.: Handling uncertainty in autonomic systems. In: Int'l Wrkshp. on Living with Uncertainty, Atlanta, Georgia (2007)
7. Esfahani, N., Kouroshfar, E., Malek, S.: Taming uncertainty in Self-Adaptive software. In: The joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Szeged, Hungary (2011)
8. Malek, S., Edwards, G., Brun, Y., Tajalli, H., Garcia, J., Krka, I., Medvidovic, N., Mikic-Rakic, M., Sukhatme, G.S.: An architecture-driven software mobility framework. Journal of Systems and Software **83** (2010) 972–989
9. Menasce, D.A., Dowdy, L.W., Almeida, V.A.: Performance by Design: Computer Capacity Planning By Example. Prentice Hall PTR (2004)
10. Weyns, D., Malek, S., Andersson, J.: FORMS: a formal reference model for self-adaptation. In: Int'l Conf. on Autonomic Computing, Washington, DC (2010) 205–214
11. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer **36** (2003) 41–50
12. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with reusable infrastructure. IEEE Computer **37** (2004) 46–54
13. Trouwborst, A.: Precautionary rights and duties of states. Martinus Nijhoff (2006)
14. Cooray, D., Malek, S., Roshandel, R., Kilgore, D.: RESISTing reliability degradation through proactive reconfiguration. In: Int'l Conf. on Automated Software Engineering, Antwerp, Belgium (2010)
15. Winkler, R.L.: Uncertainty in probabilistic risk assessment. Reliability Engineering & System Safety **54** (1996) 127132
16. Bertsekas, D.P., Tsitsiklis, J.N.: Introduction to Probability, 2nd Edition. Athena Scientific (2008)
17. Kolmogorov, A.: Foundations of Probability. (1933)
18. Hoff, P.D.: A First Course in Bayesian Statistical Methods. 2nd printing. edn. Springer (2009)
19. Zadeh, L.A.: Fuzzy sets. Information and control **8** (1965) 338353
20. Zadeh, L.A.: Fuzzy sets as a basis for a theory of possibility. Fuzzy Sets Syst. **100** (1999) 9–34
21. Coletti, G., Scozzafava, R.: Conditional probability, fuzzy sets, and possibility: a unifying view. Fuzzy Sets and Systems **144** (2004) 227–249
22. Dubois, D., Prade, H.: Possibility theory, probability theory and Multiple-Valued logics: A clarification. Annals of Mathematics and Artificial Intelligence **32** (2001) 3566 ACM ID: 590454.
23. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.: RELAX: incorporating uncertainty into the specification of Self-Adaptive systems. In: Int'l Requirements Engineering Conf., Atlanta, Georgia (2009) 79–88
24. Cheng, B.H., Sawyer, P., Bencomo, N., 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 (2009) 468–483

25. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy goals for Requirements-Driven adaptation. In: Int'l Requirements Engineering Conf., Sydney, Australia (2010) 125–134
26. Elkhodary, A., Esfahani, N., 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 (2010) 7–16
27. Poladian, V., Sousa, J.P., Garlan, D., Shaw, M.: Dynamic configuration of Resource-Aware services. In: Int'l Conf. on Software Engineering, Scotland, UK (2004) 604–613
28. Poladian, V., Garlan, D., Shaw, M., Satyanarayanan, M., Schmerl, B., Sousa, J.: Leveraging resource prediction for anticipatory dynamic configuration. In: Int'l Conf. on Self-Adaptive and Self-Organizing Systems, Boston, Massachusetts, IEEE Computer Society (2007) 214–223
29. Narayanan, D., Satyanarayanan, M.: Predictive resource management for wearable computing. In: Int'l Conf. on Mobile systems, applications and services, San Francisco, California (2003) 113128 ACM ID: 1189041.