CRATER: Case-based Reasoning Framework for Engineering an Adaptation Engine in Self-Adaptive Software Systems

Mohammed Abufouda
Computer Science Department
Technical University of Kaiserslautern
abufouda@cs.uni-kl.de

Abstract

Self-adaptation allows software systems to autonomously adjust their behavior during run-time by handling all possible operating states that violate the requirements of the managed system. This requires an adaptation engine that receives adaptation requests during the monitoring process of the managed system and responds with an automated and appropriate adaptation response. During the last decade, several engineering methods have been introduced to enable self-adaptation in software systems. However, these methods lack addressing (1) run-time uncertainty that hinders the adaptation process and (2) the performance impacts resulted from the complexity and the large number of the adaptation space. This paper presents CRATER, a framework that builds an external adaptation engine for self-adaptive software systems. The adaptation engine, which is built on Case-based Reasoning, handles the aforementioned challenges together. This paper is braced with an experiment illustrating the benefits of this framework. The experimental results shows the potential of CRATER in terms handling run-time uncertainty and adaptation remembrance that enhances the performance for large number of adaptation space.

Keywords: Self-adaptive software, Run-time Uncertainty, Case-based Reasoning, Software Performance.

I. Introduction

It is doubtless that software systems play a vital role in the modern daily activities. This creates more challenges that need to be addressed. Software engineering aims at providing software of quality by addressing these challenges and improving the existing solutions. One of these challenges is to build a *self-adaptive software system*. The majority of the existing work in the literature agrees that *self-adaptation* in software systems is the ability of a software system to adjust its behavior autonomously during run-time to handle a software system's complexity and maintenance costs as well as to preserve the system's requirements [31], [11]. This property dictates the presence of an adaptation mechanism in order to build the logic of self-adaptation. Apparently, this requires reducing the human interference as much as possible which represents a challenge in the development process of self-adaptive systems particularly when the operating states(configurations) of the managed system are relatively large. In addition, many challenges exist in the area of self-adaptive software systems that need to be tackled in order to provide an efficient and flexible adaptation process. This paper is concerned with the following challenges:

- *C1:Run-time uncertainty handling:* Uncertainty is a challenge that exists not only in self-adaptive software systems but also in the entire software engineering phases including requirements engineering, design and execution [28]. Run-time uncertainty can hinder the adaptation process if not handled and diminished efficiently. Therefore, handling uncertainty at run-time level in self-adaptive systems is an essential issue.
- *C2:Performance impacts caused by adaptation space:* The adaptation process provokes a performance challenge if the adaptation space is relatively large, particularly when new adaptations are required to be inferred. This requires an efficient mechanism that guarantees learning new adaptations as well as providing the adaptation with a satisfactory performance. Thus, the responses of the adaptation engine should be provided as soon as requested, otherwise late responses could be futile.

This paper is intended to present and validate CRATER, a framework for engineering and enabling self-adaptation in software systems. It provides an external adaptation engine[1] that reduces the changes in the managed system. CRATER benefits from Case-based Reasoning (CBR) [3] as an external adaptation engine in order to overcome the aforesaid challenges and to automate the closed control loop proposed in [1]. Specifically, CRATER provides the following contribution solutions:

- *S1:* It handles the run-time uncertainty that appears in the adaptation process due to unpredicted changes in the the environment of the managed system. This is done by incorporating the probability theory and the utility functions [29].
- *S2:* It improves the performance, namely the response time, of the adaptation process. This is done by managing the complexity of the adaptation space through remembering the previously achieved adaptations using a knowledge base. Without a knowledge base the adaptation process will be laborious for complex adaptation space.

[1]By external we mean that the logic of the adaptation is not embedded within the managed system itself.

The rest of this paper is structured as follows. Section II demonstrates a motivating example that illustrates the need of self-adaptation. Section III provides an overview of CRATER and its components. Section IV presents a detailed view about CRATER model and specification. Section V and Section VII contain details about the implementation and the results respectively. Section VIII discusses and evaluates the results while Section IX lists the related work to this research. Finally, Section X summarizes the work of this research.

II. Motivating example

The motivating example is a software system controlling a robot that requires self-adaptive behavior during run-time. This motivating example is used for both motivating the need for self-adaptive software systems and for the experimentation and the validation of CRATER. The idea of the robot is derived from [15] with an attribute extension to provide more variety of configurations. Figure 1 shows an abstract view of the robot managed system which has an exploratory task and should transmit the captured videos to a remote controlling center. Even though the example is from the robotics field, we emphasize that our concern is only the software system that manages the self-adaptive behavior of the robot rather than the robot itself. This means that the robot as a managed system could be any other system that requires enabling the self-adaptation property. We will use this example as a running example through this paper.

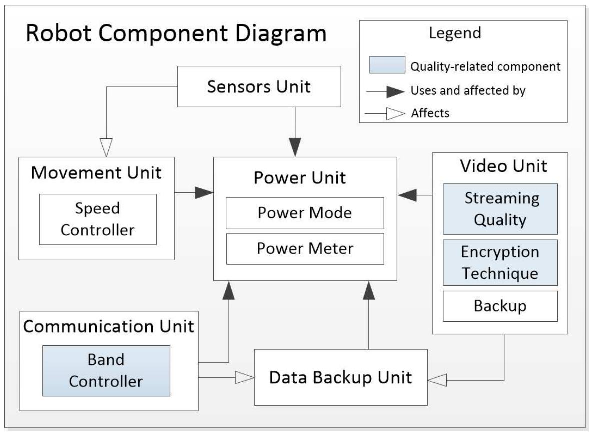


Fig. 1: Abstract view of the robot components.

The components in Figure 1 are dependent on each other; one component may affect other component(s). This dependency contributes to providing a set of various possible states of the robot, which is useful in explaining how CRATER works. The robot requires to adapt its behavior during run-time in order to keep fulfilling its requirements without manual controlling from the remote controlling center. This adaptation is a response to the changes in the environment where the robot is working and/or the changes in the attributes of the robot itself e.g. the speed and the power. These requirements include the quality of service (QoS) requirements and functionality requirements that need to be achieved by the robot self-adaptively.
An example of QoS requirements is *Video Quality* where the robot aims at keeping the quality of the transmitted video as good as possible. This is done by selecting the appropriate video quality automatically during run-time. The available power affects this requirement because higher video qualities require more power consumption than lower ones. The robot should control this process efficiently. Another example of QoS requirements is *Transmission Security* where the robot should keep the transmitted data as secure as possible during submitting it to the remote controlling center. This is achieved by selecting one among a set of encryption techniques where each technique has its advantages and drawbacks in terms of power consumption, security level, and encryption performance. An example of functionality requirements is *Robot Fitness* where the robot should manage the relations among its attributes in order to keep itself as fit as possible. For instance, the robot should reduce its speed if the power is not sufficient or an obstacle is detected by the sensors unit. Another example of functionality requirements is to enable the data backup if the communication with the remote center is lost. This requires choosing a suitable video quality due to the limitation of the space of backup storage. The challenges that the robot system may face in the self-adaptation context and are addressed by our framework automatically are:

- *Run-time uncertainty handling:* The robot may fail to identify one of its environmental variable values during its operation. For example, the sensors may fail to indicate whether there is an obstacle in the area or not(this state is a run-time uncertain state). In such problematic situations, the robot should behave tolerably; otherwise the robot may run into unwanted states.
- *Adaptation space complexity impacts:* If the robot has *N* attributes each of them has *M* different possible values, then the number possible states $\mathcal{S}$ that the robot may run in are: $\mathcal{S} = \prod_{i=1}^{N} M_i$. This requires an efficient handling of these operating states that guarantees an accepted performance. Concretely, the response time of the adaptation engine is a crucial issue because any delayed adaptation response could be useless. For example, if the robot's communication with the remote

center has been lost, then the robot should start the back-up storage in order to keep all the captured videos. Such decisions should be provided to the robot immediately; otherwise the robot could deviate from fulfilling its requirements.

III. CRATER OVERVIEW

In this section, an overview of CRATER will be presented. Based on Figure 2, which illustrates CRATER's reference model, the following subsections describe the *Managed system* and *CRATER's adaptation engine* that is decomposed into the *Adaptation mediator* and the *Case-based reasoning engine*.

A. The managed system

The managed system is the system that needs to adapt its run-time behavior autonomously e.g., the robot system discussed in Section II. In order to utilize CRATER, the managed system must provide a set of its self-adaptation concerned attributes. An example of these attributes, based on the motivating example discussed in Section II, is shown in Table I. The table also

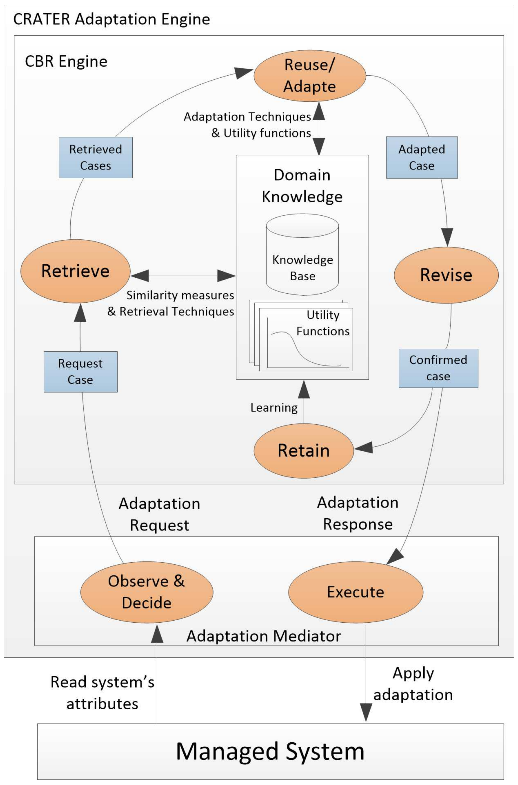


Fig. 2: CRATER Reference Model.

shows the complexity of the adaptation space size i.e., the robot may run in one of 8640 possible different configurations.

TABLE I: Robot attribute data sheet

Attribute	Values set
Communication	{OFF, VHF, Xband, UHF}
Power Mode	{Full Power, Medium Power, Saving Mode}
Power Meter	{Low, Medium, High}
Speed	{Low, Medium, High}
Video quality	{Very low, Low, Medium, High, Very High}
Data Backup	{On, Off}
Obstacles	{True, False}
Encryption	{Zig-Zag Permutation, Puer Permutation, Naive, Video Encryption Algorithm (VEA)}

B. CRATER's adaptation engine

This section provides details about the components of the adaptation engine.

1) Adaptation samples: Before digging deeper in the model's details, it is better to show how CRATER works with two adaptation samples. We assume that the managed system, the robot in our case, provides a service with a utility U and an adaptation process is issued when this utility is below or is approaching [2] a predefined utility threshold UT. Table II illustrates two randomly selected adaptations from the experiment that will be discussed later, one of them contains uncertain value. The first adaptation request, embraces a defect in the operating mode of the robot as there is an obstacle while the robot speed is high which represents a violation of the functional requirement *Robot Fitness*. The adaptation response for this unwanted state of the robot is to reduce the speed. Reducing the speed is the only possible adaptation response as we can not change the obstacle to false as it is not adaptable attribute [3]. The table shows that the utility of the adaptation request is 0.484 which is a utility threshold breaker, assuming that UT is 0.5. CRATER managed to provide an adaptation response with utility 0.892 which is greater than 0.5. The other adaptation request holds uncertain value in the communication attribute. CRATER issued an adaptation process for this robot state because the uncertain attribute, the communication, is uncertain and one possible values, *off*, leads to utility less than UT. When the communication attribute goes off, it breaks the UT, which means that the robot is unable to establish a connection with the remote center. As a result CRATER issues an adaptation process that produces the adaptation response that assures that the communication is set with appropriate value to enable communication with the remote center. Needles to say that the chosen value, UHF, should not break the utility of the robot which is satisfied and the utility is 0.8666. Another possible adaptation response for the second adaptation request is to enable the data back up and to set off the communication. However, CRATER did not choose this scenario because its utility is less than the utility of the chosen adaptation response. This is because the ultimate goal of the framework is to maximize the utility of the managed system.

TABLE II: Adaptation Samples

Attribute	**Adaptation request 1**	**Adaptation response 1**	**Adaptation request 2**	**Adaptation response 1**
Communication	UHF	UHF	**?**	**UHF**
Power Mode	Saving Mode	Saving Mode	Medium Power	Medium Power
Power Indicator	High	High	High	High
Speed	**High**	**Low**	Low	Low
Video quality	Very High	High	Low	Low
Data Backup	Off	Off	Off	Off
Obstacles	**True**	**True**	False	False
Encryption	Puer Perm.	Puer Perm.	Zig-Zag Permu.	Zig-Zag Permu.
Utility	**0.484**	**0.892**	**?**	**0.8666**

2) CRATER's adaptation mediator: Now, we present in more details the description of the framework. As shown in Figure 2, the adaptation mediator is responsible for:

- *Monitoring* the managed system by reading its attributes to decide whether an adaptation is required or not. CRATER expects that the managed system provides a service with overall utility U. The *adaptation request* is the set of the attributes' values of the managed system at the time of issuing the adaptation process. Consequently, the adaptation request is sent to the adaptation engine to start the adaptation process.
- *Executing* the adaptation response received from the adaptation engine. The adaptation response is the result of the adaptation process performed by the adaptation engine, which is the corrective state to be applied on the managed system.

3) Case-based reasoning engine: CRATER's adaptation engine is built mainly on Case-based Reasoning (CBR) which facilitates the automation process of the adaptation. CBR is an artificial intelligence technique that mimics the human behavior in solving problems based on the solutions of previous and similar problems. Generally, a case is an object that contains some attributes e.g. the robot attributes shown in Table I and, traditionally, the attributes of a case are divided into problem related attributes and solution related attributes. In our work we model the adaptation request as problem part of a CBR case and the adaptation response as solution part of a CBR case. Specifically, the red attributes in Table II represent a problem part of a CBR case and the green attributes represent the solution part of a case. The task of the framework is to find out an appropriate solution for these red attributes. Traditional CBR life cycle, as shown in Figure 3, consists of four stages:

1) *Retrieve:* The CBR system retrieves the most similar case(s) from the *Knowledge Base* by applying the similarity measures on the request case. In [33], [6], [17], many similarity measures for improved case retrieval have been introduced. Figure 4 shows an example of how the similarity is performed on the cases from the knowledge base and which attributes are considered in the similarity measures.

[2]This is because CRATER treats self-adaptation in a reactive or a proactive way depending on the implementation of monitoring process within the adaptation mediator.

[3]In Section IV-B1 we will see how CRATER classifies the attributes of the managed system.

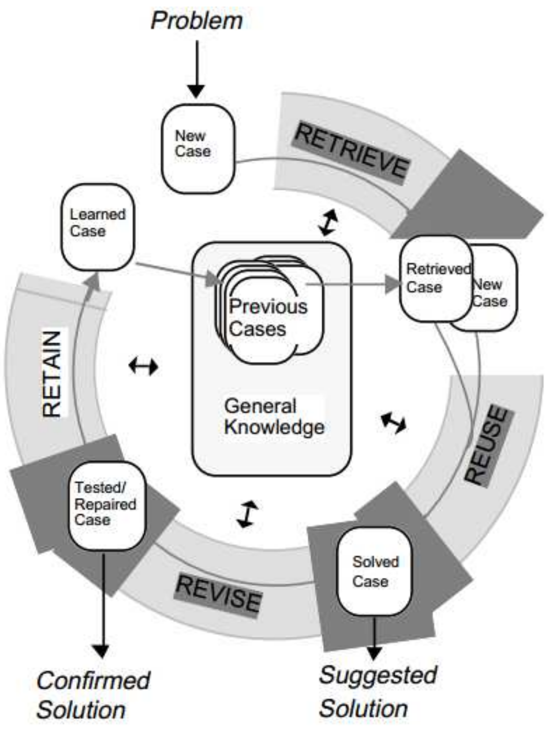


Fig. 3: Case-based Reasoning Life Cycle [3]

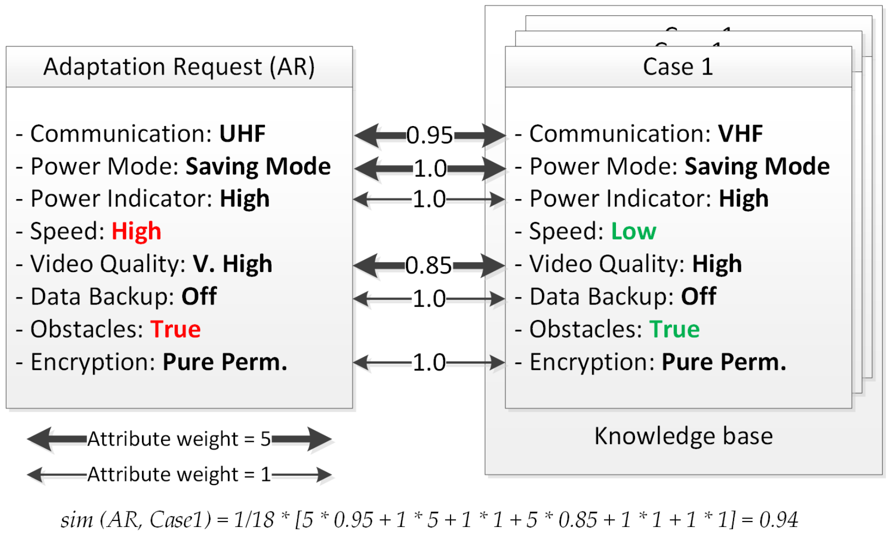


Fig. 4: An example of the similarity measure between an adaptation request and a case from the knowledge base.

2) *Reuse (Adapt):* In this stage, CBR benefits from the information of the retrieved cases. If the retrieved cases are not sufficient in themselves to solve the request case, the CBR engine adapts this/these case/s to generate a new solution. Some of the common techniques for reusing and adapting the retrieved knowledge are introduced in [39]. CRATER uses *Generative Adaptation* [27], which requires some heuristics, e.g. utility functions, to provide an efficient adaptation process.
3) *Revise:* A revision of the new solution is important to make sure that it satisfies the requirements of the managed system. The revision process can be done by applying the adaptation response to real world, evaluate it by the domain expert, or by simulation approaches. To enhance the automation of the adaptation process, we use utility functions which revise the generated adaptation and judge its utility satisfaction on the fly.
4) *Retain:* In this stage, the new generated cases are saved in the knowledge base. Case-Based Learning (CBL) have been introduced in [5] to provide algorithms and approaches for an efficient retain process.

IV. CRATER MODEL AND SPECIFICATIONS

In this section we explain how CRATER tackles the challenges described in Section I. Precisely, it explains the adaptation process and how the utility functions are used.

A. *The knowledge base*

The knowledge base in our framework contains the states of the managed system that satisfy its requirements. This property is guaranteed in the retain process where no case is retained unless it has a utility greater than the utility threshold UT. The knowledge base is modeled by the domain experts by capturing all attributes of the managed system that are related to the adaptation process. The operations performed on the knowledge base are restricted to case retrieval and case retention. Table III shows an excerpt from the knowledge base for the motivating example discussed in Section II. Assuming that the utility threshold is 0.5, it is clear from the table that all the cases in the knowledge base has a utility greater than the utility threshold.

TABLE III: Excerpt from the knowledge base.

Attribute	**Case$_1$**	**Case$_2$**	**Case$_3$**	**Case$_4$**	**Case$_5$**
Communication	UHF	VHF	VHF	UHF	UHF
Power Mode	Medium	Medium	Full	Full	Medium
Power Indicator	High	High	High	Low	High
Speed	Low	Medium	Medium	Medium	Medium
Video quality	V.Low	High	V.High	Medium	Medium
Data Backup	Off	Off	Off	On	Off
Obstacles	False	False	False	True	True
Encryption	Puer Permu.	Zig-Zag Perm.	VEA	Puer Perm.	VEA
Utility	0.813	0.603	0.758	0.565	0.928

B. *The managed system attributes*

The managed system operating states are modeled as CBR cases. Each case has a set of attributes and each attribute has a type and a weight.

1) Attribute types: Case attributes can be flagged as one or more of the types shown in Table IV. During the design of the managed system, each attribute must be labeled as adaptable or unadaptable. During the analysis process of the adaptation request, CRATER identifies UT-breaker and utility-antagonist attributes. CRATER alters the UT-breaker to provide adaptation response with utility greater than the UT. For providing an optimal adaptation response, CRATER alters the utility-antagonist attributes, which raises the utility of the provided adaptation response.

TABLE IV: Managed system attribute types.

Attribute Type	**Description**
Adaptable	An attribute whose value can be changed during the adaptation process e.g *Speed*.
Unadaptable	An attribute whose value can not be changed during the adaptation process e.g *Obstacles*.
UT-breaker	An attributes whose value participates in reaching a goal-violating state.
Utility-antagonist	An attribute whose value participates in decreasing the overall utility.

2) Attribute weights: It is normal that the attributes of the managed system vary in their effect on the utility of the provided service. Based on that, Pareto principle [26] is applied and each attribute is weighted in order to provide optimal representation of the state of the managed system.

C. *Utility functions*

Utility functions are incorporated in CRATER reference model in order to: (1) assess the cases of the knowledge base in terms of satisfying the requirements of the managed system, (2) provide a heuristic for the adaptation process and provide affirmation regarding the adaptation response expediency, (3) analyze the adaptation requests to identify UT-breaker attributes, and (4) determine when to issue the adaptation process; i.e. if the managed system's overall utility reaches or is approaching the UT.

1) Utility function definition: Utility function is a function that maps a set of attributes to a value if certain condition holds. For simplicity, the utility function definition in CRATER is based on the work in [24] and extended in order to combine multiple utility-involved attributes. The utility function is defined as in Equation 1:

$$Utility_{(a_1,...,a_i)} = \begin{cases} v_1 & if condition_1 \ holds \\ v_2 & if condition_2 \ holds \\ . & \\ . & \\ v_{n-1} & if condition_{n-1} \ holds \\ v_n & Otherwise \end{cases} \tag{1}$$

where:

- *(a_1,...,a_i)* is the set of involved managed system attributes.
- *(v_1,...,v_n)* are the values of the utility function.
- *(condition$_1$,...,condition$_{i-1}$)* is a set of condition for satisfying the utility function.

An example of the utility function is shown in Equation 2 which describes the relation among *Power Mode*, *Video Quality* and *Encryption Technique*:

$$U_{(P,Q,E)} = \begin{cases} 0.1 & if\text{(P=3 and (Q=1 or Q=2) and E=1) } holds \\ 0.5 & if\text{(P=2 and Q=2 and E=1) } holds \\ 0.8 & if\text{(P=1 and Q=3 and E=3) } holds \\ 0.99 & \text{Otherwise} \end{cases} \tag{2}$$

2) Utility functions weight: In reality, the adaptation-involved attributes of the managed system can be shared by more than one utility function due to the correlation among these attributes. Weighting these utility functions is important in modeling the managed system's requirements. The weighting process is normally the task of the domain expert and can be improved by weight learning.

3) Overall utility function: The *Weighted Geometric Mean* (WGM) is used to estimate the overall utility of the managed system in terms of its utility functions. If we have a set of utility function values $U = \{u_1, u_2, ..., u_n\}$ with corresponding weights $W = \{w_1, w_2, ..., w_n\}$, then the overall utility is estimated by the following equation:

$$U_{overall} = (\prod_{i=1}^{n} u_i^{w_i})^{1/(\sum_{i=1}^{n} w_i)} \tag{3}$$

D. CRATER's adaptation process

In this section we describe the adaptation process shown in Figure 5. The adaptation process goes through the following phases:

1) Analyzing adaptation request: When CRATER's adaptation engine receives an adaptation request, it analyzes it to identify the attributes that breaks UT and the attributes that antagonize the managed system utility. This identification process is done by comparing the adaptation request values to the utility functions. That is, any attribute that participates in making any of these utility function to be below the UT is considered as utility breaker attribute. Similarly, any attribute that decreases any of the utility functions is considered as antagonistic attribute. The identification of these attributes helps in providing efficient adaptation responses by changing the values of these two types of attributes to get higher utility from the adaptation response.

2) Case retrieval: Case retrieval is a CBR core functionality. CRATER retrieves the most similar case(s), if exist, to the *request case* as shown in Figure 2. It is important to mention that the request case is formulated from the adaptation request by excluding the UT-breaker attributes from it. This exclusion is inevitable as the knowledge base keeps only cases of best operating states that have no UT-breaker values at all. After this formulation of the request case, it is ready for the similarity measure calculation, as shown in Figure 4, to find its best matching cases. For example, if the robot system is running with the following attributes: {*Power:= Full power, Video Quality:= Very High, Obstacles:=True, Speed:= Fast*}, then, obviously, this state represents an unwanted operating state because the robot speed should not be fast if an obstacle is detected. This state is a typical adaptation request as it represents a deviation from the system requirements that are defined by a utility function similar to Equation 2. As the speed is the UT-breaker attribute in this example, the request case will be formulated by excluding it from the adaptation request. So the adaptation case of that adaption request is: {*Power:= Full power, Video Quality:= Very High, Obstacles:=True*}.

3) Constructing qualified adaptation frame (QAF): The retrieval process returns a set of cases called *Qualified Adaptation Frame* QAF, such that each case C_k in this set satisfies the condition:

$$\beta \leq Sim(C_k, Adap_{Req}) \leq 1 \tag{4}$$

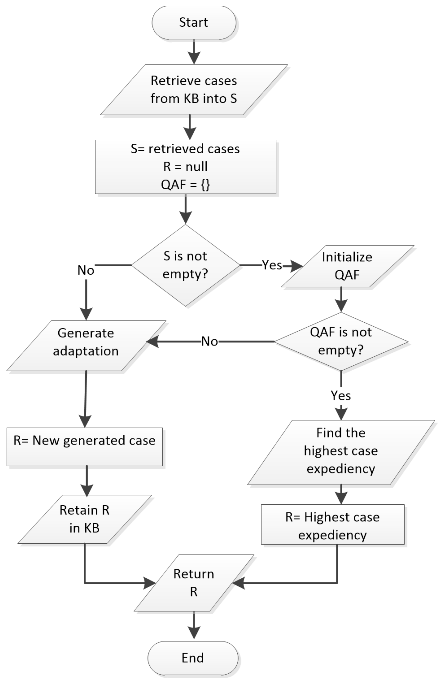


Fig. 5: Adaptation process flow chart.

where β is a value between [0,1] and represents the minimal similarity value for accepting retrieved cases from the knowledge base and Sim is a function that calculates the similarity between the adaptation case and each of the retrieved case. Having the QAF ready, a decision on which adaptation response to select has to be taken. In fact, similarity is not the only decisive factor, however, the utility of the retrieved case is also considered. This combination is called *Case Usefulness*. CRATER calculates the usefulness of a case in the QAF by Equation 5:

$$CU(c)_{QAF} = 1 - [(1 - sim(Adap_{req}, c)) \cdot utility(c)] \tag{5}$$

where *CU* is the *Case Usefulness* for each case c in the QAF and sim is the similarity between the adaptation request $Adap_{req}$ and the case c. This combination in calculating case usefulness is essential. On the one hand, the inclusion of similarity of the retrieved cases in calculating case usefulness is important as higher similarity leads to fewer changes in the managed system attributes. On the other hand, the inclusion of the utility reflects the quality of the case in terms of meeting the managed system's requirements.

4) Generating adaptation response: If the QAF is empty, CRATER generates the adaptation response based on the utility function by adapting the request case attributes in order to provide a case with a utility greater than UT. This process is called *Utility-guided constructive adaptation* which has two flavors. (1) *First Fit Heuristic*: which is a normal iterative search process in the space values of the attributes that is applied on the request case [27]. The first value that causes the utility of adaptation request to be greater than UT is returned as an adaptation response. (2) *Best Fit Heuristic*: which is an extension of the first fit heuristic with extra capability; that is the search process finds values that maximize the utility of the adaptation response (i.e. providing an optimal adaptation). If the adaptation response is generated by one of the previous ways, the utility of the generated case is considered as the case usefulness.

5) Retaining: Retain phase is restricted to the newly generated adaptation response from the Utility-guided constructive adaptation process. As all of the generated adaptation responses have a utility greater than UT, they are qualified for retention in the knowledge base for future reuse.

It is clear that CRATER is able to start operating with an empty knowledge base, which enables a full automation of the adaptation process. The utility functions govern the learning process, which guarantees the quality of retained cases. The number of the retained cases in the knowledge base decreases overtime which raises the likelihood of retrieving the adaptation response instead of generating it. This has a positive impact on the performance of CRATER and reduces the response time of the adaptation engine significantly. Algorithm 1 abstracts the automation of adaptation process of our solution.

E. Run-time uncertainty diminution in CRATER

CRATER's ultimate goal is to provide an adaptation response that maximizes the utility of the managed system. Therefore, when the managed system is running under uncertain state, consequently its utility is not deterministic. In this case CRATER

Algorithm 1 Adaptation process

```
Require: $KB$ , $A_{req}$
Ensure: $Utility(A_{res}) > UT$
 1: List cases $\Leftarrow$ Retrieve ($KB$,$A_{req}$)
 2: List QAF
 3: Case $A_{res}$
 4: while Case c $\Leftarrow$ Iterate(cases) do
 5:     if Sim($A_{req}$,c) $\in [\beta,1]$ then
 6:         QAF.add(c)
 7:     end if
 8: end while
 9: if QAF is not Empty then
10:     $A_{res} \Leftarrow$ max(CaseUsefulness(QAF) )
11:     Return $A_{res}$
12: else
13:     $A_{res} \Leftarrow$ ConstructiveAdapt($A_{req}$)
14:     Retain($A_{res}$,KB)
15: end if
16: Return $A_{res}$
```

needs to quantify this uncertainty to provide efficient adaptation responses. To that end, we are identifying uncertainty by capturing its three dimensions [37]:

1) the *Location* of uncertainty: uncertainty is revealed within CRATER's model in two locations. The first location (Location 1) is the managed system state and second location (Location 2) is within the QAF.
2) the *Nature* of the uncertainty in Location 1 is the run-time uncertainty which is the knowledge shortage in the managed system attributes' values. This could be due to environmental reasons or measurement errors [4] in providing known values. The nature of uncertainty in Location 2 is the variability such that the QAF has more than one case with the same maximum highest usefulness.
3) the *Level* of uncertainty needs to be estimated. Otherwise, CRATER will not be able to decide whether an adaptation is required or not. To estimate the level of uncertainty in Location 1, CRATER starts with generating a set κ of all possible states that the uncertain state can be one of them. Then, the number $\Re$ of states that belongs to κ and require adaptation is calculated. Subsequently, the probability μ that the uncertain state is a UT-breaker is determined as seen in Equation 6. Also, the uncertainty degree in the managed system Θ is estimated as shown in Equation 6.

$$\mu = \frac{\Re}{Size(\kappa)},\ \Theta = \frac{\#uncertain\ attributes}{\#all\ state\ attributes} \tag{6}$$

Finally, the overall uncertainty *Level* η is estimated by Equation 7.

$$\eta = 1 - [(1-\mu)\cdot(1-\Theta)] \tag{7}$$

Even though this work handles the run-time uncertainty in Location 1, by calculating the uncertainty level, the framework provides a naive solution for estimating the level of the uncertainty in Location 2 (due to variability in the QAF). This solution do not require any further calculations. That is, if there is more than one case with the same highest usefulness in the QAF, then the selected case is the case with the highest utility as it satisfies the ultimate goal of CRATER we mentioned earlier in this section.

In the context of utility functions, there are two ways to deal with uncertainty: (1) *Optimistic Paradigm:* which deals with the uncertain values as values that heighten the utility and (2) *Pessimistic Paradigm:* which deals with the uncertain values as values that belittle the utility. Both pessimistic and optimistic paradigms are not preferable in systems like the one in our motivating example. This is because when the robot is operating in an optimistic paradigm and its uncertain state dictates an adaptation, the optimistic paradigm will fail to issue an adaptation process. Likewise, when the robot is operating in a pessimistic paradigm and its uncertain state do not dictate an adaptation, the pessimistic paradigm will cause performance overhead for issuing useless adaptation.

To diminish the run-time uncertainty efficiently, we introduce a *Hybrid Paradigm* that depends on a cutoff value, $\eta_{threshold}$ [5]. If $\eta_{threshold}$ is *one* then it behaves pessimistically i.e. an adaptation process is issued whenever the managed system runs in uncertain state, and when $\eta_{threshold}$ is zero, CRATER behaves optimistically i.e. no adaptation process is issued. Intuitively,

[4]For example sensor or actuator errors and problems.

[5]This value is defined during the configurations of CRATER

$\eta_{threshold}$ should maintain a value greater than zero and less than one. An adaptation process is issued only when η is less than or equal $\eta_{threshold}$.

V. Implementation

In order to validate CRATER, a prototypical implementation of our framework has been developed. FreeCBR [2] engine is used as CBR infrastructure for the adaptation engine that provides the retrieval functionality. Java Development Kit (JDK) version 1.7 is used for developing the remaining parts including the retain and the adapt/reuse components. Also an implementation of the the first fit heuristic was done in our work. For the robot system, we implement a simple simulation software that provides the set of the attributes of the robot during run-time. This simulator is used as a basis for providing the adaptation requests. That means, the simulator randomly forces the robot to run into a state the requires an adaptation by randomly altering its attributes values. The monitoring component [6], which we implement also, keeps reading the robots attributes to detect whether an adaptation is required or not by automatic calculation of robots run-time utility. Similarly, the simulator was used to randomly inject uncertain values in the run-time state attributes of the robot. We implement that by adding *UNKNOWN* value to the value lists of all attributes. Afterwords, when the simulator assign a random value to each attribute, *UNKNOWN* value will eventually appear in the status of the robot which represents uncertainty.

VI. Experiment

Figure 6 shows our paradigm to fulfill a mature experimentation and validation. The following sections will implement the processes shown in that figure. In this section we will provide details about the conducted experiment including the settings, the runs and the research hypotheses that we validate.

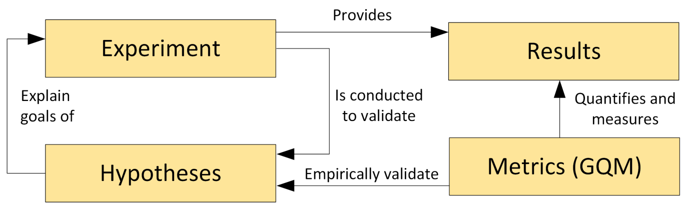


Fig. 6: Experimentation and validation process.

A. Experiment settings

In order to validate CRATER, a simulation-based experiment is conducted based on the motivating example described in Section II and the implemented software described in Section V. The experiment was performed under Windows 8 (x64) machine with 4 GB of RAM and CPU Intel CORE 2 Duo (P7750) 2.26 GHz. The experiment requires configuring the parameters of the framework as following: (1) UT is 0.5, which gives the chance to show the framework's ability in providing adaptation with greater utility, (2) β is 90%, which is suitable to show how CRATER retrieves and constructs cases, (3) First fit heuristic is used in the implementation of the prototype, (4) Starting with an empty knowledge base which enhances the confidence of the results [7], and (5) $\eta_{thresold}$ is 85%, which is fair enough to show how CRATER reacts under uncertainty. These settings are used strictly throughout the experiment. However, some additional changes are provided in Section VII to show the effects of changing these parameters.

B. Experiment runs

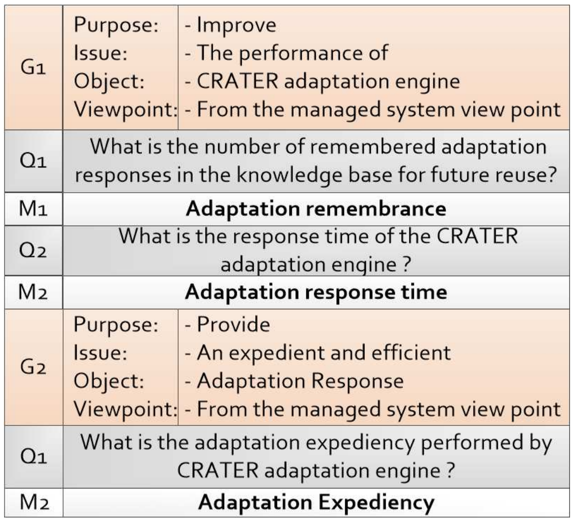
In order to perform the experiment, CRATER is subjected to seven successive runs, and each run contains 50 adaptation requests. We decided to have seven runs instead of one long run to provide a detailed information about how our framework is working particularly and to show the effect of having a knowledge base. We chose 50 adaptation request per run, which means 350 total runs, because it is sufficient to provide all important results that we need. That is after the 350 run the results become more stable in terms of response time. Unless otherwise stated, the runs embraces no uncertainty.

C. Software metrics

In order to provide an empirical evidence regarding our research, we hypothesize our claims. To do so, we will provide an evaluation metrics that will help during the evaluation of the framework. Adaptation-related metrics are formulated using GQM approach [35]. Based on Figure 7, the metrics will be used during the validation process are shown in Equations 10, 9 and 8.

[6]The implementation of this component behaves in a reactive way, i.e., the adaptation is issued when the robot is running in unwanted state.

[7]If we started with non-empty knowledge base, then our experiment will be biased. This is why we decided to start with empty knowledge base



G1	Purpose:	- Improve
	Issue:	- The performance of
	Object:	- CRATER adaptation engine
	Viewpoint:	- From the managed system view point
Q1	What is the number of remembered adaptation responses in the knowledge base for future reuse?	
M1	**Adaptation remembrance**	
Q2	What is the response time of the CRATER adaptation engine ?	
M2	**Adaptation response time**	
G2	Purpose:	- Provide
	Issue:	- An expedient and efficient
	Object:	- Adaptation Response
	Viewpoint:	- From the managed system view point
Q1	What is the adaptation expediency performed by CRATER adaptation engine ?	
M2	**Adaptation Expediency**	

Fig. 7: GQM Sheet for the validation of our framework.

$$Adaptation\ Expediency = \frac{\#Expedient\ Adaptations}{\#All\ Adaptations} \tag{8}$$

$$Average\ Response\ Time = \frac{\sum_{i=1}^{n} Response\ Time(n)}{\#All\ Adaptations} \tag{9}$$

$$Adaptation\ remembrance = \frac{\#Adaptations\ retrived\ from\ KB}{\#All\ Adaptations} \tag{10}$$

D. Research Hypotheses

In this section we will present our three hypotheses and link each of them to the corresponding metric for validation purpose.

1) Adaptation Expediency: It is important to show that the framework provides a useful adaptation whenever it is required. This work innovates a quality metric, named: *Adaptation Expediency* for that sake. An expedient adaptation is the adaptation that rescues the managed system from its unwanted state; i.e. an adaptation response with utility greater than the UT. If an expedient adaptation response is always provided, then the adaptation expediency in Equation 8 equals one. However, in some cases the managed system's resources (e.g. power unit) decrease overtime which affects the overall utility of the managed system. CRATER has nothing to do in this case as it has no authority on the unadaptable attributes of the managed system. Instead, an adaptation response with the highest possible utility is provided.

H1: *CRATER is in position to provide an expedient adaptation response whenever an adaptation is issued.* To validate this hypothesis, software metric, *Adaptation Expediency* shown in Equation 8, is used later on.

2) Handling complex adaptation space impacts: As stated earlier, complex adaptation space affects the response time of the adaptation engine. In this work we want to empirically prove that CRATER is able to provide an efficient adaptation.

H2: *CRATER is in position to handle the performance impacts caused by the complexity of the adaptation space.*

To validate this hypothesis, a performance metric, *Response Time* shown in Equation 9, is used later on. In addition, a new metric, *Adaptation Remembrance* shown in Equation 10, is used to shows the effect of utilizing a knowledge base and its vital role in reducing the response time.

3) Handling Run-time uncertainty: The third hypothesis of our experiment is about run-time uncertainty handling.

H3: *CRATER is in position to provide an expedient adaptation response under run-time uncertainty.*

To validate this hypothesis, we will use the same software metric, *Adaptation Expediency* shown in Equation 8, to test the expediency of the adaptation under run-time uncertainty.

VII. Results and interpretation

This section provides an extensive view on the results from the discussed experiment in Section VI and in the light of the three hypotheses discussed in Section VI-D. The results will be mapped to each of the hypotheses in Section VI-D.

A. Adaptation expediency

Figure 8 shows a box-plot diagram for the adaptation expediency. CRATER showed success in providing an expedient adaptation, i.e. an adaptation response with utility greater than UT (0.5). The variation of the value of the provided adaptation is due to the variation of the adaptation requests per se. The variation of the adaptation requests is guaranteed by the pure random generation of the adaptation requests.

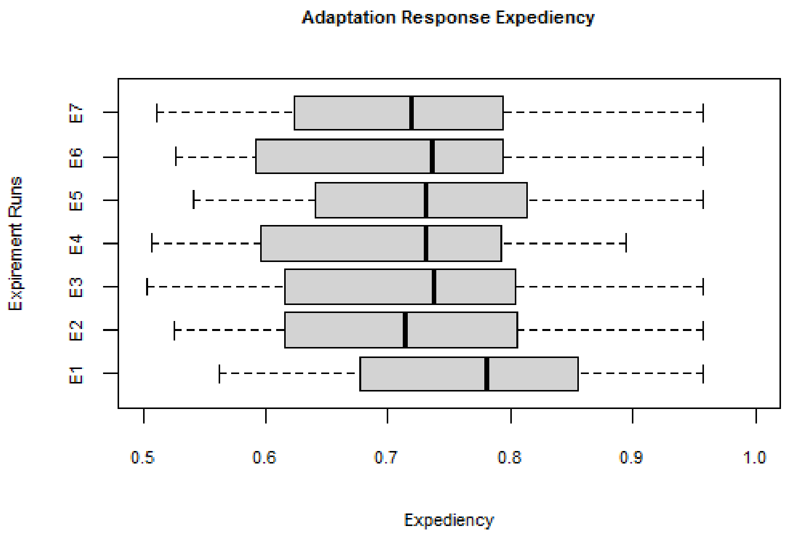


Fig. 8: Adaptation expediency for the adaptation requests of the 7 runs (each run has 50 adaptation request).

B. Adaptation response time

Figure 9 shows the average response time for the conducted experiment runs with and without the usage of a knowledge base. It is clear that the response time decreases over time when using the knowledge base. This is clear from the figure as it

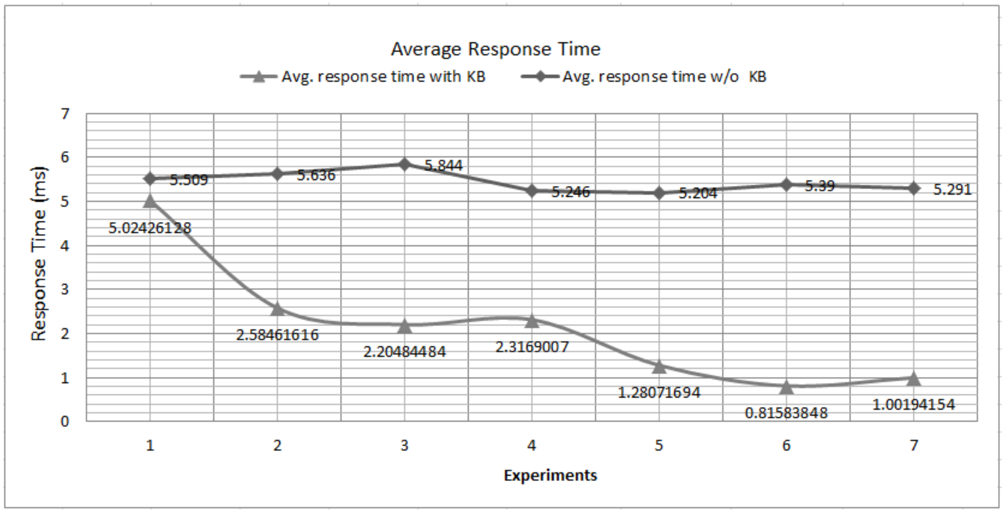
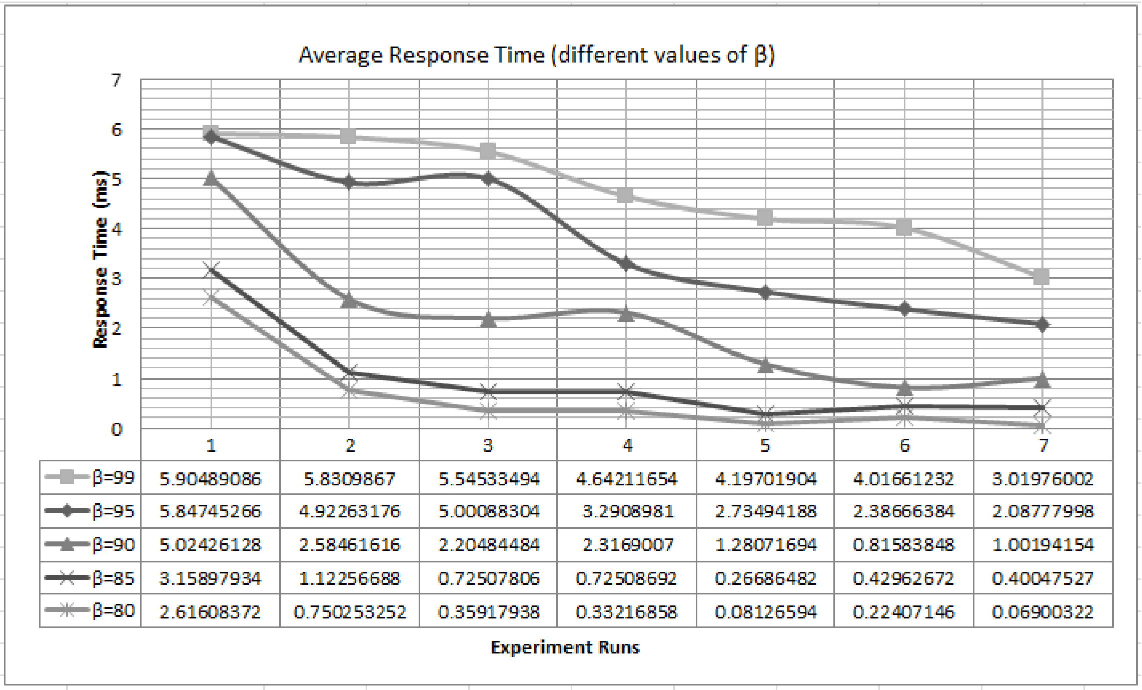


Fig. 9: Adaptation response time for the adaptation requests of the 7 runs (each run has 50 adaptation request). The figure shows the effect of having the knowledge base in the framework.

shows that not using a knowledge base keeps the response time roughly constant with its heights value. We included the results without a knowledge base just to show its positive impact on the response time. We simulate the absence of the knowledge base by providing the adaptation responses by construction, and not by retrieving them from the knowledge base. When using the knowledge base, the average response time for each run of the experiment is greater than the average response time for

the subsequent run. That is because the adaptation responses that were generated and not retrieved from knowledge base were in the early experiment runs. In other words, the later run's average response time decreases because the adaptation responses began to be retrieved form the knowledge base which consumes less time than constructing them.
Figure 10 shows the impact of the value of β on the average response time. If the value of β is high (e.g. 99%,) then fewer cases are selected from the knowledge base into the QAF, which leads to more constructively generated adaptation responses and, consequently, higher response time.

Figure 11 shows the average response time for adaptation requests under uncertainty. The figure shows that the response time



	1	2	3	4	5	6	7
■ β=99	5.90489086	5.8309867	5.54533494	4.64211654	4.19701904	4.01661232	3.01976002
◆ β=95	5.84745266	4.92263176	5.00088304	3.2908981	2.73494188	2.38666384	2.08777998
▲ β=90	5.02426128	2.58461616	2.20484484	2.3169007	1.28071694	0.81583848	1.00194154
✕ β=85	3.15897934	1.12256688	0.72507806	0.72508692	0.26686482	0.42962672	0.40047527
✳ β=80	2.61608372	0.750253252	0.35917938	0.33216858	0.08126594	0.22407146	0.06900322

Fig. 10: The effect of choosing different values of β on the response time.

decreases overtime for the different runs of the experiment for the same reason described earlier. Figure 11 shows that there is

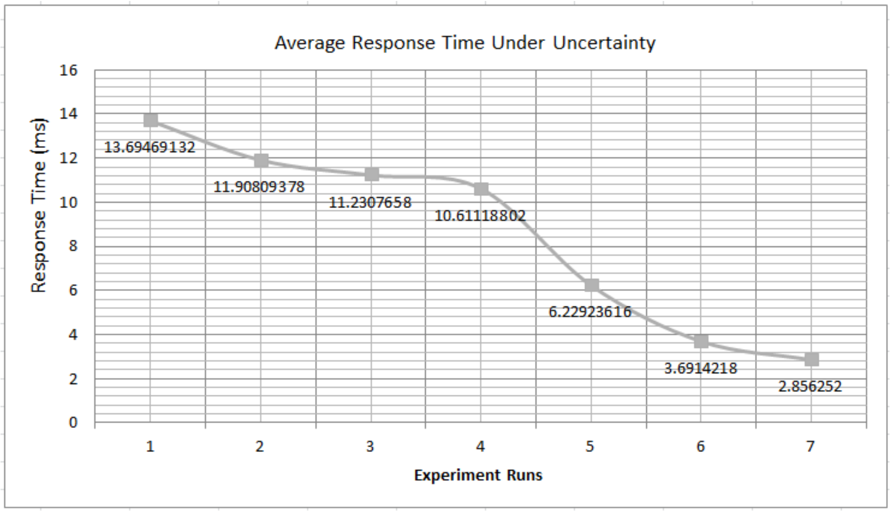
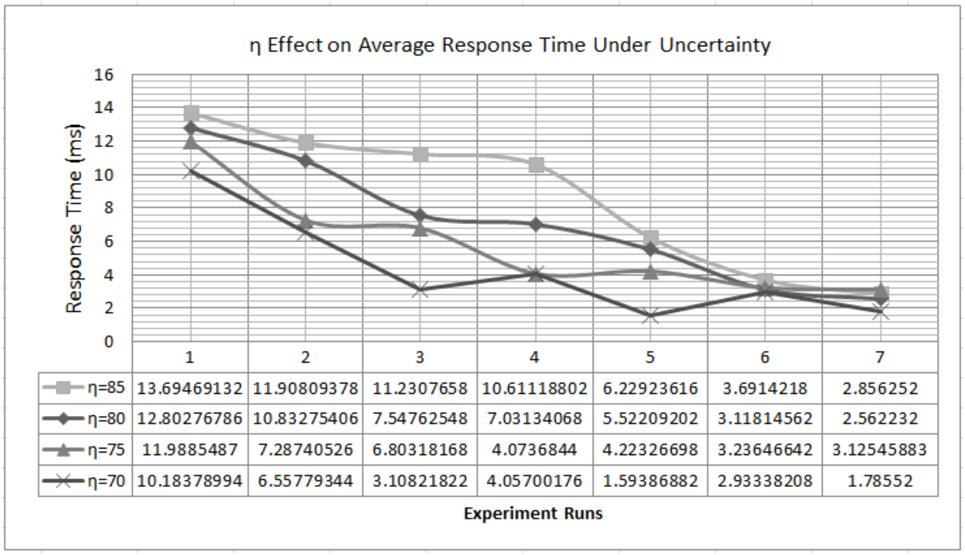


Fig. 11: The response time under uncertainty for 7 runs each of them has 50 adaptation request.

an increase in the average response time compared to what appears in Figure 9. This is normal because the adaptation requests with uncertain values require more processing and analysis to estimate η as described in Section IV-E. Figure 12 shows the average response time for adaptation request under uncertainty with different values of $\eta_{thresold}$. As seen, $\eta_{thresold}$ affects the performance such that the less $\eta_{thresold}$ is the less the average response time is. This is because when CRATER is configured



	1	2	3	4	5	6	7
■ η=85	13.69469132	11.90809378	11.2307658	10.61118802	6.22923616	3.6914218	2.856252
◆ η=80	12.80276786	10.83275406	7.54762548	7.03134068	5.52209202	3.11814562	2.562232
▲ η=75	11.9885487	7.28740526	6.80318168	4.0736844	4.22326698	3.23646642	3.12545883
× η=70	10.18378994	6.55779344	3.10821822	4.05700176	1.59386882	2.93338208	1.78552

Fig. 12: The effect of choosing different values of $\eta_{thresold}$ on the response time.

with a high value of $\eta_{threshold}$, then more uncertain states of the managed system are being considered as adaptation requests. The remembrance metric finds the relation between the retrieved adaptation from the knowledge base and the total number of adaptations. Figure 13 shows the remembrance rate of adaptation responses through the experiment runs. In the first run, the number of constructed adaptation responses is more than the retrieved, which is intuitive as the knowledge base is empty at that point. In later runs of the experiment, the number of retrieved adaptation began to increase unlike the number of constructed adaptation responses. This provides a positive impact on the adaptation response time and it is the main reason behind the decreasing response time over runs.

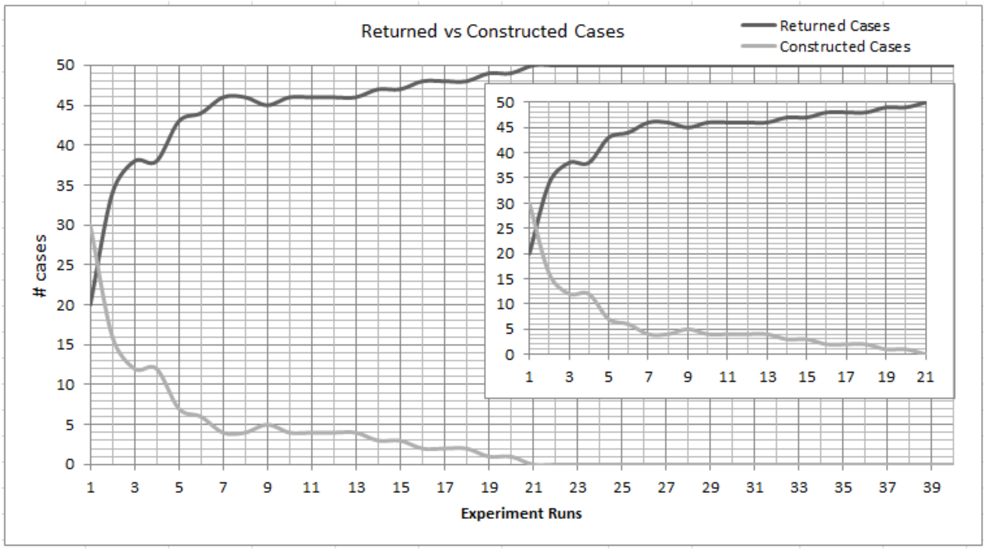
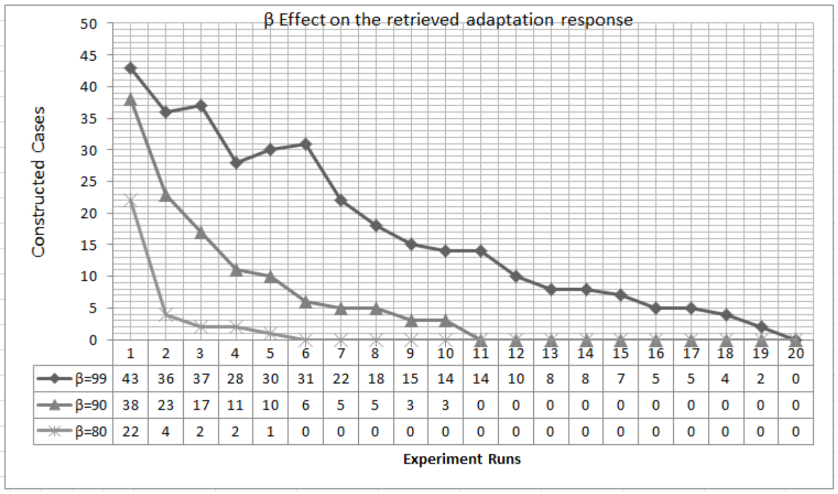


Fig. 13: The number of retrieved cases vs the number of the constructed cases of the experiment for 40 runs.

Figure 14 shows how β affects the number of constructed adaptation. It is clear from the figure that the more β value is the more constructed adaptation responses are.

C. Run-time uncertainty

Figure 15 shows the expediency of the adaptation process for adaptation requests that contain uncertain values in the run-time state. It is clear that CRATER manages to work under uncertain situation and provides an efficient adaptation in terms of adaptation expediency. We notice that the average adaptation expediency under uncertainty, in Figure 15, is more than the average adaptation expediency without run-time uncertainty, in Figure 8. The interpretation is that, when there is uncertain



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
β=99	43	36	37	28	30	31	22	18	15	14	14	10	8	8	7	5	5	4	2	0
β=90	38	23	17	11	10	6	5	5	3	3	0	0	0	0	0	0	0	0	0	0
β=80	22	4	2	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 14: The correlation between the parameter β and the number of the constructed cases.

attribute value in the state of the managed system then many values are considered for this uncertain attribute. This makes CRATER relaxed to choose the value that heighten the utility, and hence the expediency, of the adaptation response.

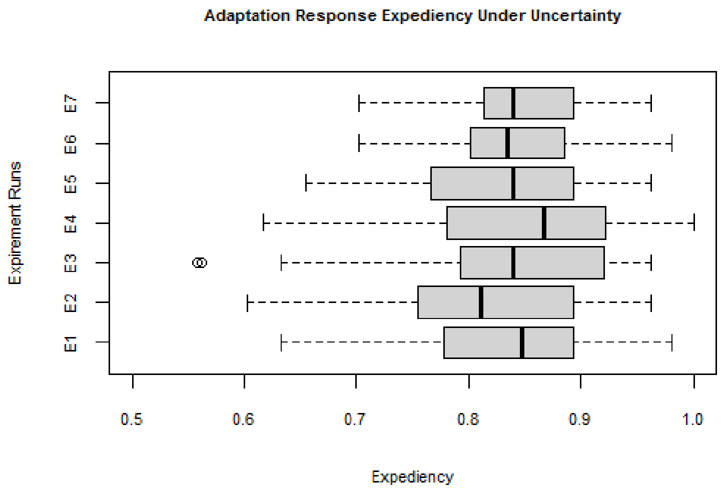


Fig. 15: Adaptation expediency under run-time uncertainty for the 7 runs.

VIII. Evaluation and discussion

In order to provide empirically evaluated evidence regarding the potential of our framework, we will validate the hypotheses presented in Section VI-D by mapping each hypothesis to empirical values obtained from Section VII.

A. *CRATER's adaptation expediency*

As shown in Figure 8, it is obvious that our framework is in position to provide an expedient adaptations whenever it is asked to. From that figure we conclude that CRATER provides an expedient adaptation response with an average *0.7216* which validates the first hypothesis **H1**.

B. *CRATER's adaptation response time*

The remembrance rate of the cases as shown in Figure 13 increases overtime which enables CRATER to reuse cases stored in the knowledge base. This is clear from the figure as the average returned cases from the knowledge base is *46.85* cases versus *3.15* constructed cases out of *50* cases. This result means that, in the conducted experiment, CRATER provides *93.7%* of its

adaptation responses from the knowledge base and the rest of the adaptation responses,*6.3%* , were generated constructively. This affects the performance positively as constructing new adaptation responses consumes more time than retrieving it from the knowledge base.
Based on Figure 9, the response time decreases from *5.02 ms* in the first run of the experiment to *1.01 ms* in the last run of the experiment. The average response time the performed seven runs of the experiment is *2.175 ms*. The average response time under uncertainty based on Figure 11 is *8.6 ms*. This means that our framework succeeded in enabling self-adaptation with better performance, which validates the second hypothesis **H2**.

C. CRATER's run-time uncertainty handling

Uncertainty handling in CRATER is validated by identifying the adaptation expediency under run-time uncertainty. Based on results shown in Figure 15, the average adaptation expediency of the performed runs of the experiment is *0.834*, which represents an efficient adaptation under uncertainty knowing that the utility threshold is 0.5 and the maximum utility of the managed system is 1.0. This validates the third hypothesis **H3**.
Based on the previously discussed evaluation, it is clear that CRATER provides an effective mechanism to overcome the impacts of complex adaptation space. It memorizes the previously performed adaptation for later use which improves the performance of the adaptation engine. In addition, it can operate under uncertainty which is an advantage over traditional solutions.

D. Experiment Validity

The *internal threats to validity* in the conducted experiment include: (1) The adaptation requests were generated randomly to represent a diversity of adaptation requests. The randomness of generation was guaranteed by the pure random selection of attribute values in the implemented adaptation request generator component of the prototypical implementation of CRATER. (2) Different CBR implementations and similarity measures could provide slightly different results, particularity in terms of response time even. However, the chosen CBR implementation [2] showed acceptable performance. The *external threats to validity* or the generalization of the results could be affected by the chosen domain. This could be figured out if CRATER is utilized in a different domains. However, no major differences are expected in the results.

E. Limitations

CRATER could suffer if the managed system's adaptation related attributes do not have a predefined possible values. CRATER's mechanism is built upon a defined set of attributes values. In addition, the attributes should be in a discrete form, however, if an attribute has a continuous domain we overcome this problem by discretizing its values. An example of this kind of attributes is the speed as in the running example. We discretize the speed to three different values: Slow, Medium and High.
Learning approaches do not come for free. Online and offline learning always requires some training time. In our work, as shown in Figure 16, the response time was high at the beginning of the experiment runs then it gets decreasing before being stable after the seven run. This is why we cut our results in section Section VII to seven runs. The figure shows that it takes *15.23 ms* before the response time settles to the most possible minimum value without uncertainty and *60.22 ms* under uncertainty. We should mention that this overhead is for 8640 possible operating states, and the overhead will increase by increasing the number these states. The prototypical implementation may be improved to provide better performance particularly if the CBR

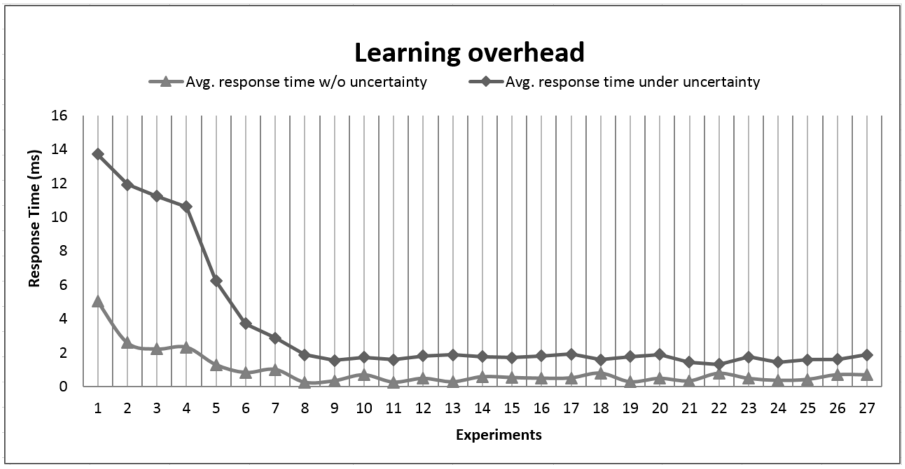


Fig. 16: Learning overhead.

implementation is enhanced with different representation of the knowledge base to enable some methods of cashing instead of reading the whole knowledge base every time. The experiment used to validate our framework used 8 attributes for the robot system. A performance degradation could exist if the managed system has a relatively huge number of attributes with more possible values. This could affect the performance of CRATER. However, an offline indexing for the knowledge base can be performed to enhance the performance of the retrieval process which is not used in the implemented version of CRATER in this paper.

IX. Related work

Self-adaptive software systems have been studied for several years from different perspectives tying to address several challenges [31], [11] that are managed system dependent. This is why the area of self-adaptive software system has a plethora of frameworks and approaches for enabling self-adaptation in different domains at different development levels. This leads in the difficulty in providing a universal framework that fits for every system. Consequently, the comparison between these frameworks is not an easy task and sometimes it is not viable at all for many reasons: (1) the variation of application domains hinders a sound comparison, i.e. self-adaptation in information systems concentrates on realizing self-adaptation in middleware [9] and service oriented architectures [10], [19] levels and (2) the variation of the covered aspect of software engineering process itself, e.g. the uncertainty handing in our work, which is run-time uncertainty, differs from the uncertainty handling in [15], which is the design-time uncertainty at the architecture level. In addition, the maturity of some highly related work [30], [20], [36], [32] hampers a sound comparison due to the lack or absence of experimentation and empirical data.
Our work, which has been published in short versions in [4], is in the area of software engineering and it is a step towards *systematic engineering of self-adaptive system* that aims at: (1) providing *better performance by remembering* the previously achieved adaptations and (2) handling the *run-time uncertainty*. We will present the related work based on these facts. After investigating the recent work related to our research, we found that ours differs significantly from the existing work in that, it handles the run-time uncertainty. To our knowledge, run-time uncertainty has not been addressed before. Beside run-time uncertainty, handling the performance impacts caused by the complexity of the adaptation space is an additional contribution that makes our work new to the field. Based on that, we find a number of methods and frameworks that are related to what we did. We will classify and discuss them in the following subsections along with a clarification of their merits and possible drawbacks in the light of our framework's contribution.

A. *Performance and adaptation space complexity in self-adaptive software systems:*

The self-adaptive software engineering research area is rich in the number of frameworks that *engineers* a self-adaptive software system. In this section we will provide the related frameworks and connect them to our first contribution, the performance and adaptation space complexity handling. In [30], [20], [14], a learning approaches are utilized to enable self-adaptation by capturing the the managed system's requirement and goals. Even though the work in [30], [14] incorporates a knowledge base in their framework, but the impacts of this incorporation was not discussed and validated in terms of system performance. On the other side, we discussed empirically how the inclusion of a knowledge base affects the performance. The work [20] lacks the performance impacts evaluation caused by using online learning of their reinforcement learning solution. Similar to FUSION [14], the work in [18] utilized a feature-based representation to provide a dynamic adaptation that supports non-functional requirements. An advantage of this work over FUSION is that it addresses, like CRATER, the complexity of the adaptation space. However, both FUSION [14] and [18] lack uncertainty handling. GRAF [13], MOCAS [7] and [36] were proposed for engineering self-adaptive software systems based on different representations of the architectures of the systems at run-time. Such kind of approaches has a performance overhead because they reproduces a new adaptable version of the managed system components which affects the performance particularly at run-time. Adapta framework [32] was presented as a middleware that enabled self-adaptation for components in distributed applications. The monitoring service in Adapta monitored both hardware and software changes. The work in [40] proposed a fuzzy-based self-adaptive software framework. This framework has a set of design steps in order to realize the adaptation. The overhead in these design steps and their level of automation represent a limitation in this approach as they hinders the automation process in enabling self-adaptation. Also, the overhead of fuzzy nature impacts of the approach was not covered. Even though the works in [30], [7], [36], [32], [40] are interesting, they lack experimentation and empirical evaluation of the framework. Morin et al. [25] presented an architectural-based approach for realizing software adaptation using model-driven and aspect-oriented techniques. The aim of this approach was to reduce the complexities of the system by providing architectural adaptation solution. The evaluation of the work in [25] was limited to check whether the framework provides the required adaptation, or configuration as they call it, or not. A noticeable limitation of this work is that it is only able to cope with restricted number of adaptation variability, unlike CRATER which aims at operating under huge number of possible configurations. RAINBOW [16] monitors the managed system using abstract architectural models to detect any constraints violation and then provides architectural adaptation template. The authors evaluate RAINBOW in terms of the performance of the managed system, which was web-based client-server, which showed a noticeable improvement. One limitation of RAINBOW [16] is that the adaptation styles are predefined, unlike CRATER, which increases management costs of the entire self-adaptive system particularly when we have a large number of adaptation space. Tajalli et al. [34] proposed PLASMA, a plan-based and architecture-based mechanisms

for enabling the self-adaptation. Basically, PLASMA provides an adaptive layered architecture that contains planners which create planes for both the application and adaptation layers. One drawback of this work is that, it is not able to select the best configuration, unlike CRATER thanks to utility functions, as they do not incorporate the quality properties of the configuration. Also, there was a performance issues in the construction of the configurations. In [9], a prototype for seat adaptation was provided using a middleware to support an adaptive behavior. This approach was restricted to the seat adaptation which is controlled by a software system. One limitation of this approach is that the adaptation space is static and hard-coded in the system. MOSES [10] provided self-adaptation for service oriented architecture systems. The authors used linear programming problem for formulating and solving the adaptation problem as a model-based framework. The work in [21] introduced a configuration-based self-adaptive mechanism that considers the quality of alternative configuration, however, it do not provide a real-time evaluation of there algorithms. Accord [23] is a programming framework that facilitates realizing self-adaptation in self-managed applications. The configuration policies for this framework was predefined which increases the complexity of the system in case of having large number of configurations.
With a satisfactory confidence, we can believe that our work provides an advantage over all of the previous related work in that: *it handles the run-time uncertainty*. All of the previous work lacks dealing with uncertainty at any of its levels.

B. Uncertainty in self-adaptive software systems

Some of the related work considers uncertainty. In this section will shed light on these works and relate them to our work. An interesting framework, POISED [15], introduced a probabilistic approach for handling uncertainty in self-adaptive software systems by providing positive and negative impacts of uncertainty. This work addressed the uncertainty that appears during the design time and due to architecture variation. This work do not incorporate a knowledge base which affects the performance of their approach as the whole number of the possible configuration is considered every time.
RELAX [38] and [11], [22] handled and analyzed uncertainty at the requirement levels for different domains. RELAX [38] framework handled uncertainty at the requirement level of the self-adaptive system by providing a new requirements language for self adaptive systems. The work in [12] builds on RELAX [38] specification language to specify more flexible requirements to handle the uncertainty at requirement level. Similarly, [8] provided a solution to requirements uncertainty by introducing a requirement reflections framework. In [22], uncertainty analysis at the requirement level was provided for a service-oriented self-adaptation.
From uncertainty perspective, our work differs from the previous related work in that: our work tackles the run-time uncertainty in self-adaptive software systems.

X. Conclusion and future directions

In this paper, CRATER has been presented as a framework for constructing an external adaptation engine for realizing self-adaptation in software systems. This framework benefits from case-based reasoning as an adaptation engine along with the utility functions in order to provide efficient adaptation responses to be applied on the managed system. In this research, we showed how our framework utilizes a knowledge base to improves the performance(response time) by remembering the previously performed adaptations. Additionally, it has successfully been able to tackle the run-time uncertainty that hinders the adaptation process. A prototypical implementation of CRATER has been presented and experiments have been conducted with an empirical evaluation for the results. In the future work, CRATER will be applied on different domains e.g. information system, in order to unequivocally validate its applicability. Another possible improvement of our implementation is to provide adaptation proactively. This means that CRATER can be extended to predict the status of the managed system so that when it is approaching the utility threshold, then an adaptation should be issued.

References

[1] An architectural blueprint for autonomic computing. http://www-03.ibm.com/autonomic/pdfs/ACBlueprintWhitePaperV7.pdf/.

[2] Freecbr engine. http://freecbr.sourceforge.net/.

[3] Agnar Aamodt and Enric Plaza. Case-based reasoning; foundational issues, methodological variations, and system approaches. *AI COMMUNICATIONS*, 7(1):39–59, 1994.

[4] Mohammed Abufouda. A framework for enhancing performance and handling run-time uncertainty in self-adaptive systems. *International Journal of Computer Science & Information Technology*, 6(1), 2014.

[5] David W. Aha. Case-based learning algorithms. Proc. 1991 DARPA Case-Based Reasoning Workshop. Morgan Kaufmann, 1991.

[6] Selim Aksoy and Robert M. Haralick. Probabilistic vs. geometric similarity measures for image retrieval. In *IEEE Conf. Computer Vision and Pattern Recognition*, 2000.

[7] C. Ballagny, N. Hameurlain, and F. Barbier. Mocas: A state-based component model for self-adaptation. In *Self-Adaptive and Self-Organizing Systems, 2009. SASO '09. Third IEEE International Conference on*, pages 206–215, 2009.

[8] N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, and E. Letier. Requirements reflection: requirements as runtime entities. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, pages 199–202, 2010.

[9] G.M. Bertolotti, A. Cristiani, R. Lombardi, M. Ribaric and, N. Tomas andevic and, and M. Stanojevic and. Self-adaptive prototype for seat adaption. In *SASOW Fourth IEEE International Conference*, 2010.

[10] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Francesco Lo Presti, and Raffaela Mirandola. Qos-driven runtime adaptation of service oriented architectures. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 131–140.

[11] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems. chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, 2009.

[12] Betty H. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 468–483, 2009.

[13] Mahdi Derakhshanmanesh, Mehdi Amoui, Greg O'Grady, Jürgen Ebert, and Ladan Tahvildari. Graf: graph-based runtime adaptation framework. SEAMS '11, pages 128–137.

[14] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the 18 ACM SIGSOFT international symposium*, FSE '10, pages 7–16, 2010.

[15] Naeem Esfahani. A framework for managing uncertainty in self-adaptive software systems. In *26th IEEE/ACM International Conference on Automated Software Engineering*, pages 646–650, 2011.

[16] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, pages 46 – 54, 2004.

[17] Ping Guo, Qian Bao, and Qian Yin. Probabilistic similarity measures analysis for remote sensing image retrieval. *Machine Learning and Cybernetics, 2006 International Conference*, 2006.

[18] Hisayuki Horikoshi, Hiroyuki Nakagawa, Yasuyuki Tahara, and Akihiko Ohsuga. Dynamic reconfiguration in self-adaptive systems considering non-functional properties. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1144–1150, 2012.

[19] Mohamed-Hedi Karray, Chirine Ghedira, and Zakaria Maamar. Towards a self-healing approach to sustain web services reliability. In *AINA Workshops'11*, pages 267–272, 2011.

[20] Dongsun Kim and Sooyong Park. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *SEAMS '09. ICSE Workshop on*, pages 76 –85, 2009.

[21] Benjamin Klopper et al. Planning with utility and state trajectory constraints in self-healing automotive systems. In *Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference on*, pages 74–83, 2010.

[22] Jingsheng Lei, FuLee Wang, Mo Li, and Yuan Luo. Requirement uncertainty analysis for service-oriented self-adaptation software. In *Network Computing and Information Security*, volume 345 of *Communications in Computer and Information Science*, pages 156–163. 2012.

[23] Hua Liu, Manish Parashar, and Senior Member. Accord: A programming framework for autonomic applications. *IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Engineering Autonomic Systems*, 36:341–352, 2006.

[24] Daniel A. Menascé, John M. Ewing, Hassan Gomaa, Sam Malex, and João P. Sousa. A framework for utility-based service oriented design in sassy. In *Proceedings of WOSP/SIPEW '10*, pages 27–36. ACM, 2010.

[25] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44 –51, October 2009.

[26] Vilfredo Pareto. Cours d'economie politique. *F. Rouge, Lausanne*, 1896.

[27] Enric Plaza and Josep Lluís Arcos. Constructive adaptation. In *Proceedings of the 6th European Conference on Advances in Case-Based Reasoning*, ECCBR '02, pages 306–320, 2002.

[28] A.J. Ramirez, A.C. Jensen, and B. H C Cheng. A taxonomy of uncertainty for dynamically adaptive systems. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 99–108, 2012.

[29] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[30] Mazeiar Salehie and Ladan Tahvildari. A quality-driven approach to enable decision-making in self-adaptive software. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 103–104. IEEE Computer Society, 2007.

[31] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2009.

[32] Marcio Augusto Sekeff Sallem and Francisco Jose da Silva e Silva. Adapta: a framework for dynamic reconfiguration of distributed applications. In *Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM '06)*, 2006.

[33] Armin Stahl. *Learning of Knowledge-Intensive Similarity Measures in Case-Based Reasoning*. PhD thesis, 2003.

[34] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic. Plasma: a plan-based layered architecture for software model-driven adaptation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 467–476, New York, NY, USA, 2010. ACM.

[35] Rini van Solingen, Vic Basili, Gianluigi Caldiera, and H. Dieter Rombach. Goal question metric (gqm) approach. In *Encyclopedia of Software Engineering*. John Wiley and Sons, Inc., 2002.

[36] Thomas Vogel and Holger Giese. Adaptation and abstract runtime models. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 39–48, 2010.

[37] W.E. Walker, P Harremoes, J Rotmans, JP Van der Sluijs, MBA Van Asselt, P Janssen, and MP Krayer Von Krauss. Defining uncertainty: a conceptual basis for uncertainty management in model-based decision support. *Integrated Assessment*, 2003.

[38] J. Whittle, P. Sawyer, N. Bencomo, B. H C Cheng, and J. Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *Requirements Engineering Conference, 2009. RE '09. 17th IEEE International*, pages 79–88, 2009.

[39] Wolfgang Wilke and Ralph Bergmann. Techniques and knowledge used for adaptation during case-based problem solving. IEA/AIE '98, pages 497–506. Springer-Verlag, 1998.

[40] Qiliang Yang, Jian Lü, Juelong Li, Xiaoxing Ma, Wei Song, and Yang Zou. Toward a fuzzy control-based approach to design of self-adaptive software. In *Proce. of the 2nd Asia-Pacific Symposium on Internetware*, pages 15:1–15:4, 2010.