CRATER: Case-based Reasoning Framework for Constructing An Adaptation Engine in Self-Adaptive Software Systems

by
Mohammed A. Abufouda

A thesis submitted to the
Department of Computer Science
in partial fulfilment of the requirements for the degree of
Master's of *Computer Science*

Department of *Computer Science*
Technical University of Kaiserslautern

October 2012

Abstract

Self-adaptivity in software systems is a property that allows software systems to autonomously adjust their behaviour during run-time to keep satisfying system's goal. This adjustment is performed with minimal or without human intervention. The motivation behind developing self-adaptive software systems is to enable the managed system to run independently which reduces the cost of complexities caused by manual handling of maintenance. Efficient self-adaptive software system should handle all possible operating scenarios that violate system's goals and react to these violations properly. This requires Adaptation Engine that receives adaptation request during the monitoring process of the managed system and responds with the appropriate adaptation response. In this thesis CRATER is presented which is a framework for building an external adaptation engine for self-adaptive software systems that is built on Case-Based Reasoning (CBR). CBR is an artificial intelligence branch constructed mainly on the idea of "similar problems have similar solutions". CRATER handles two of the challenging problems in self-adaptive software system: (1) Handling uncertainty that hinders the adaptation process and (2) Managing the adaptation space complexity efficiently. CRATER provides an effective mechanism for producing applicable adaptation response with effective handling for both aforementioned challenges. This thesis presents an experiment illustrating how CRATER is utilized along with results and evaluation that show a significant potential for response time, adaptation expediency and adaptation under uncertainty.

Acknowledgements

I would like to express my sincere gratitude all those who supported me during my master study and master thesis. Especially Jun. Prof. Lars Grunske who patiently spares no efforts in helping me with continuous support, advices and invaluable feedbacks during my last year. Warm thanks are dedicated to my father and mother who supported me during the whole period of my studies. A special warm thank is devoted for my wife Hadeel who faithfully heartens me during my study.

Declaration

I hereby declare that this master thesis contains no material which has been accepted for the award of any other degree or diploma in any university or equivalent institution except where due reference is made. To the best of my knowledge and belief, this master thesis contains no material previously published or written by another person, except where due reference is made in the text of the master thesis.

Kaiserslautern, September 29, 2012

Mohammed Abufouda

Contents

Abstract **ii**

Acknowledgements **iii**

Declaration **iv**

List of Tables **ix**

List of Figures **x**

Algorithms **xi**

1 Introduction **1**

1.1 Goals and contributions of this thesis 2
1.2 Research method 3
1.3 Terms definition 4
1.4 Thesis outline 5

2 Preliminaries **6**

2.1 Self-adaptive software systems 6
2.1.1 Adaptation classifications 7
2.2 Uncertainty 9
2.2.1 Uncertainty in software engineering 9
2.2.2 Uncertainty in self-adaptive software systems 10
2.3 Case-based Reasoning 11
2.3.1 CBR overview 11
2.3.2 CBR life cycle $(RE)^4$ 11
2.3.3 Similarity measures 13

2.3.4 Case retrieval 14
2.3.5 Case adaptation 15
2.3.6 Learning in CBR 15
2.4 Summary 16

3 State of The Art **17**
3.1 Related work selection criteria 18
3.2 Related work 18
3.2.1 Learning based adaptation 18
3.2.2 Architecture and model based adaptation 19
3.2.3 Middleware based adaptation 21
3.2.4 Fuzzy control based adaptation 22
3.2.5 Programming framework based adaptation 22
3.3 Discussion 23
3.4 Problem statement 24
3.5 Summary 26

4 Proposed Solution **27**
4.1 Motivating example 27
4.1.1 Robot system 27
4.1.2 Robot goals 30
4.1.2.1 Quality-related requirements 30
4.1.2.2 Functioning requirements 31
4.2 CBR knowledge base 32
4.3 Managed system attributes 32
4.3.1 Attribute types 32
4.3.2 Attribute weight 33
4.4 CBR as adaptation engine 34
4.4.1 Adaptation request 34
4.4.2 Adaptation response 36
4.4.3 Adaptation process 36
4.4.3.1 Analysing adaptation request 36
4.4.3.2 Case retrieval 38
4.4.3.3 Constructing QAF 38
4.4.3.4 Generate adaptation response 40
4.4.3.5 Retain 40

4.5 Utility function . . . 42
4.5.1 Utility function importance . . . 42
4.5.2 Utility function definition . . . 42
4.5.3 Utility function weight . . . 44
4.5.4 Overall utility function . . . 44
4.5.5 Utility function examples . . . 45
4.6 Uncertainty diminution in CRATER . . . 46
4.6.1 Uncertainty handling . . . 47
4.6.2 CRATER's uncertainty location . . . 48
4.6.3 CRATER's uncertainty level . . . 48
4.6.3.1 Adaptation request uncertainty . . . 48
4.6.3.2 Uncertainty in qualified adaptation frame . . . 50
4.7 Summary . . . 51

5 Implementation 52
5.1 CRATER architecture . . . 52
5.1.1 Knowledge base modelling . . . 52
5.1.2 CBR engine . . . 53
5.1.2.1 Adapt/Reuse component . . . 53
5.1.2.2 Retain component . . . 55
5.2 Adaptation request . . . 55
5.3 Monitoring . . . 55
5.4 Main classes . . . 55
5.5 Development tools . . . 56
5.6 Uncertainty . . . 57
5.7 Summary . . . 58

6 Experiment and Results 59
6.1 Experiment setup . . . 59
6.1.1 Design decisions . . . 59
6.1.2 Experiment nature . . . 60
6.2 GQM-based metrics . . . 60
6.2.1 Adaptation engine performance . . . 60
6.2.1.1 Adaptation remembrance . . . 60
6.2.1.2 Adaptation response time . . . 61
6.2.2 Adaptation expediency . . . 61

6.3 Results 62
6.3.1 Examples of adaptation 63
6.3.2 Response time results 63
6.3.2.1 β value effect on response time 65
6.3.2.2 Response time under uncertainty 65
6.3.2.3 η value effect on response time 66
6.3.3 Adaptation remembrance 66
6.3.4 Adaptation expediency 67
6.3.5 Results discussion and research evidence 69
6.3.6 Results conclusion 71
6.4 Experiment validity 71
6.4.1 Internal validity 72
6.4.2 External validity 72
6.5 Summary 72

7 Conclusions **73**
7.1 CRATER merits and limitations 74
7.1.1 CRATER merits 74
7.1.2 CRATER limitations 75
7.2 Prospective and vision 76

Bibliography **77**

List of Tables

3.1 Summary of Related Work 25

4.1 Robot attribute data sheet 29
4.2 Robot quality requirements 30
4.3 Encryption techniques characteristics 31
4.4 Robot functioning requirements 31
4.5 Managed System Attribute Types 33

6.1 Adaptation Samples 64

List of Figures

1.1 Self-adaptive Software System with External Adaptation Engine . . . 2

2.1 Self-adaptation Process . . . 7
2.2 Case-based Reasoning Life Cycle . . . 12

4.1 Abstract View of Robot Components . . . 28
4.2 CRATER Reference Model . . . 35
4.3 Adaptation Response Estimation Flow Chart . . . 43

5.1 CRATER's Architecture: Execution View . . . 53
5.2 Knowledge Base Snippet . . . 54
5.3 Main classes of CRATER . . . 57
5.4 Uncertainty handling classes . . . 58

6.1 GQM Sheet for Adaptation Performance Goal . . . 61
6.2 GQM Sheet for Adaptation Expediency Goal . . . 62
6.3 Average Response Time . . . 64
6.4 Average Response Time: Different β values . . . 65
6.5 Average Response Time Under Uncertainty . . . 66
6.6 Average Response Time Under Uncertainty: Different η values . . . 67
6.7 Adaptation Response Remembrance . . . 68
6.8 β Effect on the Adaptation Process . . . 68
6.9 Adaptation Expediency . . . 69
6.10 Adaptation Expediency Under Uncertainty . . . 70

List of Algorithms

1 CRATER adaptation process 37
2 First Fit Heuristic Constructive Adaptation 41
3 Best Fit Heuristic Constructive Adaptation 41
4 Estimating μ 50

Chapter 1

Introduction

Needless to say that software plays a vital role in the modern daily activity which creates more challenges that are needed to be solved. Software engineering aims to provide software of quality. One of the challenges is to construct a software with the ability of autonomous behaving during its run-time. Hence *self-adaptivity* in software systems is the ability of a software system to adjust its behavior during run-time to preserve system's goals. This property dictates the presence of adaptation mechanism in order to do the logic of self-adaptivity. Many solutions, including techniques, approaches and frameworks, exist in literature and practice to realize self-adaptivity. Self-adaptivity is required to handle software system's complexity and costs [34] and enable these systems to run autonomously. This requires reducing the human interference as much as possible which represents a challenge in the development process of self-adaptive systems particularity when the operating states and configurations of the managed system are relatively big. Many challenges exist in the area of self-adaptive software system and many contributions exist in the literature to handle these challenges. However these contributions lack the flexible and dynamic adaptation from three perspectives:

- *Adaptation responsible unit:* There is no clear separation between the managed system, the system intended to behave adaptively, and the adaptation engine. This increases the complexity and maintenance of the development of self-adaptive software system and limits the transferability of the work done for one application to another applications and domains.
- *Uncertainty handling:* Uncertainty is a challenge that exists not only in self-adaptive software system but also in the entire software engineering areas.

Therefore handling uncertainty is considered crucial issue in constructing self-adaptive software system as uncertainty hinders the adaptation process if it is not handled and diminished.

- *Adaptation space:* The corner stone of self-adaptive software system is to provide an adaptation to be applied on the managed system during run time. The adaptation process raises a performance challenge if the adaptation space is relatively big particularly in the software systems where new adaptations are required to be inferred. This challenge requires an efficient mechanism in dealing with this adaptation space that guarantees learning new adaptation along with providing efficient adaptation at satisfactory performance.

1.1 Goals and contributions of this thesis

This thesis is intended to develop and validate CRATER, a framework for an external adaptation engine for self-adaptive software system, which solves the aforementioned problems together. Concisely, in addition to provide the functionality of self-adaptive software systems, CRATER is used to (1) handle uncertainty that appears in the adaptation process which hinders the efficiency of the adaptation process, (2) enhance the performance of adaptation process particularly performance problems due to the big number of operation states and configurations and (3) provide reasonable response time of the adaptation engine which affects the whole adaptation process positively. Figure 1.1 shows an abstracted view of CRATER. The managed system is

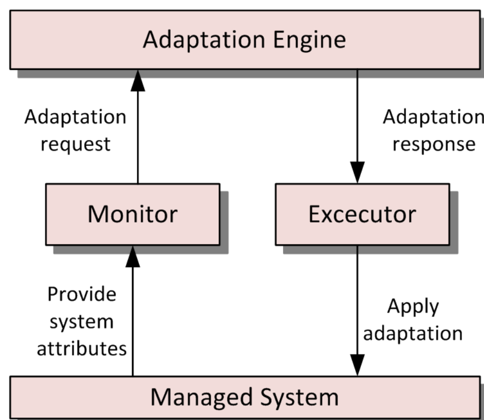


Figure 1.1: Self-adaptive Software System with External Adaptation Engine

a system that intended to upgrade to a self-adaptive software system and is separated from the external adaptation engine. CRATER requires monitoring the managed system behaviour so that it can detect any deviation from system's goals. In case that the monitor detects a violation in the managed system's goals, an adaptation request is issued and passed to the adaptation engine which is modelled in this thesis as Case-based Reasoning (CBR) engine. The adaptation engine responds to this adaptation request with appropriate adaptation response that rescues the system from its violating state to another state that keeps satisfying system's goal. Executor task's is to apply the adaptation response on the managed system.

1.2 Research method

This thesis aims to provide a trustworthy empirically validated external adaptation engine that can be utilized in the realization of self-adaptive software system. The outcome of this research should overcome the problems in the existing solutions in realizing self-adaptivity in software systems. In order to accomplish this mission, the research in this thesis follows the following research method steps:

1. *Problem statement:* The first step in this research is to identify the problem this thesis is solving. This task is carried out by literature review of the existing solution and inspecting their limitation. The outcome of this step is a clear and specific problem statement.

2. *Solution idea:* During this step a solution idea is synthesized in order to construct an external adaptation engine that overcome the problems defined in the problem statement. This step involves an implementation of the solution idea i.e. the proposed external adaptation engine in order to test and validate it.

3. *Experimental evaluation:* This step aims to validate the external adaptation engine resulted from the solution idea step of the research method. The validation of the proposed solution is performed through an empirical evaluation style. The empirical evaluation is conducted as experiment with binary validation paradigm on a motivating example illustrating the usefulness of this solution. The results of this steps provide the software engineering empirical evidence for the validation of this research.

1.3 Terms definition

In this section I will list the terms used in the following chapters with their definitions within the scope of this thesis:

- *Adaptation Engine:* Based on the work in [34], the adaptation engine is the component or the set of components that are responsible for providing self-adaptivity mechanism. Generally, an adaptation engine is an implementation of a closed control loop [15].

- *Adaptation Request:* Is an object that contains managed system's attributes values at the time when the managed system violates a certain predefined requirement.

- *Adaptation Response:* Adaptation response is the result of the adaptation process performed by the adaptation engine. Adaptation response is an object that contains the corrective state that has to be applied on the managed system.

- *Uncertainty:* Any object that has unknown values or undefined values is an object with uncertainty. An example of uncertainty is an adaptation request that has one or more attributes with unknown or undefined values.

- *Adaptation Space:* Is the set of all possible states that the managed system can run in.

- *System Goal:* Is a functional or non functional goal of the managed system.

- *Adaptable Attributes:* Are the managed system's attributes whose values can be changed during the adaptation process.

- *Qualified Adaptation Frame:* Is the set of retrieved cases from the knowledge base after the retrieving process.

- *Uncertain Value:* Any value that are not in the set of defined values for some attribute.

- *Utility-guided Constructive Adaptation:* It is the process of constructing new adaptations if the qualified adaptation frame is empty. The utility function is the decisive criteria in this process.

- *Managed System Uncertainty (Uncertain State):* Is a state of the managed system where one or more of its attributes has/have unknown values.
- *Utility Threshold (UT):* Is the utility value at which the managed system requires adaptation i.e. the utility of the managed system should not reach the value of utility threshold otherwise adaptation is issued.

1.4 Thesis outline

The rest of this thesis is organized as follow: Chapter 2 provides the relative information for the reader to gain general understanding of the context of this thesis. It also contains the used technology for the proposed solution. This chapter includes information about: (1) self-adaptive software systems, (2) uncertainty in self-adaptive software systems and (3) case-based reasoning. In Chapter 3, I will shed light on some related work and how researchers solve self-adaptivity challenges accompany with extensive evaluation for their work and relate it to CRATER. This chapter also define the problem statement of this thesis. Chapter 4 contains details about the proposed solution. It also contains a motivating example used for explanation and for the experiment. Chapter 5 is dedicated for the implementation of CRATER. Chapter 6 describes the experiment settings used to evaluate CRATER. It also contains evidences regarding this research by elaborating software measures and metrics for empirical evaluation purpose. Chapter 7 provides a conclusion of the thesis and discusses the merits of CRATER and prospective.

Chapter 2

Preliminaries

In this chapter I will pave the way for the reader to gain enough information about the topic of the thesis by providing preliminary concepts used throughout this thesis. In Section 2.1 I will start with self-adaptive software systems then provide information about uncertainty in self-adaptive software systems in Section 2.2. In Section 2.3 I will end this chapter by providing basics of Case-Based Reasoning (CBR).

2.1 Self-adaptive software systems

Software quality is considered the basic motivation in software engineering field. In order to provide a software system with an accepted quality, some quality attributes should be preserved. Software adaptability is a quality attribute that contributes in reducing the cost of handling the complexities of software systems [34] and reduces the required amount of maintenance which reduces the human involvement. Most of the work in the area of self-adaptive software engineering agrees that self-adaptive software systems are *systems that can change its behaviour during runtime while preserving software system's goals.* So as to develop a self-adaptive system, many questions should be answered for example, "When", "How" and "What" to be adapt [34]. Answering these questions is the main challenge during the development of a self-adaptive software system and characterizes the behaviour of the system and the interaction among its components.
Generally, adaptation mechanism goes through four processes [34] as shown in Figure 2.1:

i. Monitoring: In this process, a monitoring process is kept in order to perform a

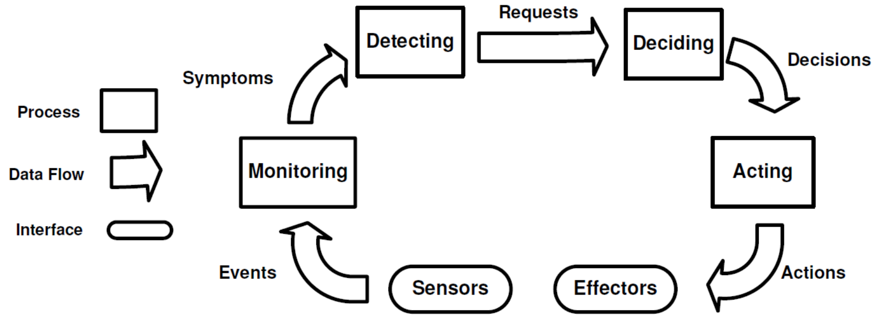


Figure 2.1: Self-adaptation Process [34]

continuous reading for system's characteristics. Generally, these characteristics are the attributes and parameters of the system that the adaptation process is build based on their values that at some point of operating necessitates adaptation.

ii. *Detecting*: Detecting process investigates and evaluates system characteristics in order to decide whether adaptation is need or not.

iii. *Deciding*: If adaptation is required, this process characterizes the adaptation nature. This includes answers to questions like (1) what system's characteristics should be changed? (2) what the nature of this change is ?

iv *Acting*: After the deciding process, an action should take place to apply the adaptation. This includes changing of the runtime behaviour of the managed system by applying the proposed adaptation.

2.1.1 Adaptation classifications

Self-adaptive software systems can be classified based on many perspectives. In the following paragraphs I will provide some of these categorizations.

1. *Adaptation responsible:* In self-adaptive software systems there are two types of adaptation based on which component or software is responsible for doing the adaptation:

   i. *External Adaptation*: In this approach similar to Figure 1.1, the adaptation engine is separated from the adaptable software system itself. The

adaptation engine provides the adaptation response once required and applies it on the managed system. This separation between the managed system and the adaptation engine enhances the scalability and maintainability of the entire system. Also, the external adaptation engine can serve multiple self-adaptive components and legacy systems that need to be adaptable.

ii. *Internal Adaptation*: In this approach, the adaptation logic is embedded in the software system itself. This approach hinders the scalability and increases the system's complexity.

2. *Adaptation response nature:* Adaptation response has two style in terms of the way they are generated:

   i. *Static Adaptation:* In this type, adaptation responses are static and the logic of adaptation is restricted to choose one of these decisions. It is obvious that this type hampers the existence of new adaptations; however, it guarantees the suitability and correctness of adaptation decisions. The work in [9] is an example of static adaptation.

   ii. *Dynamic Adaptation:* In this type, new adaptations can be evolved during run-time. Unquestionably these adaptations should guarantee the satisfaction of system's requirements which is an elementary challenge in self-adaptive software systems. In this type of adaptation, a learning process is required to learn and provide new adaptation responses as efficient reaction for new situations the managed system may run through.

3. *Adaptation process initiation:* This categorization is based on the moment at which the system issues an adaptation request during the runtime of the self-adaptive software system. The categorization includes:

   i. *Reactive Adaptation*: When the system reaches an unwanted state, then the adaptation request is issued.

   ii. *Proactive Adaptation*: The adaptation request is issued before the system reaches unwanted state. This requires a component for early detection for states that violate system's goals.

   i. *Preventive Adaptation*: In this case the fault is repaired before a consequence appears to the user [26].

In all cases monitoring system's states is required however in the proactive adaptation, more effort should be harnessed for handling the prediction and detection of unwanted state before reaching them. It is obvious that proactive adaptation can solve many drawbacks of the reactive adaptation as it precludes the system from operating in unwanted states.

2.2 Uncertainty

In this section I will provide some information about uncertainty. In Subsection 2.2.1 I will discuss some information about uncertainty in the domain of software engineering then in Subsection 2.2.2 I will present the related definitions of uncertainty in self-adaptive software systems.

2.2.1 Uncertainty in software engineering

Many challenges exist in the software engineering field, one of them is dealing with *Uncertainty*. Diminution of uncertainty becomes more and more essential; because a system running under uncertainty could raise the percentage of undesired results. Uncertainty may exist in all phases of software engineering life cycle. This means that uncertainty may appear in requirements engineering, system design and even in coding and software testing [46]. I will shed light on the general definitions and classifications of uncertainty in software engineering then I will talk about uncertainty in the self-adaptive software systems and the scope of uncertainty that will be covered in this paper. Many definitions for uncertainty exists in literature, one general definition is "*Any departure from the unachievable ideal of complete determinism*" [41]. Another definition is " *Uncertainty Principle in Software Engineering (UPSE), which states that uncertainty is inherent and inevitable in software development processes and products*" [46]. Based on these definitions I can say that the definition of uncertainty is context-specific which means that uncertainty in requirements is different from the uncertainty in system models even though the requirements and system models are related. Hence, dealing with uncertainty is different among system development phases. Also it is clear that if uncertainty exists in one development phase, the subsequent phases will inherit this property unless it is deterministically resolved.

2.2.2 Uncertainty in self-adaptive software systems

In self-adaptive software systems, uncertainty is a crucial challenge. This is because the behaviour of the system during run-time will be determined by the system itself. So, the system should behave correctly and should not dissent the functional and non-functional requirements after the adaptation has been performed. In self-adaptive software systems, the possibility of uncertainties may increase as the adaptation engine decisions will face some uncertainties in both reading the system's parameters and in judging the right adaptation decision.
Based on [41] uncertainty has three dimensions:

1. The *Location* of uncertainty: Where the uncertainty manifests in the system.

2. The *Level* of uncertainty: A variation between deterministic level and total ignorance. This means that uncertainty about one attribute of the system can take a value between one and zero [32].

3. The *Nature* of uncertainty: Whether the cause of uncertainty is variability or lack of knowledge in the uncertainty meant attribute of the system.

Uncertainty in self-adaptive software systems falls into two categories [17]:

i. *Internal Uncertainty*: This type of uncertainty is a consequence of internal models of the system and adaptation engine. This means that uncertainty is resulted from the system itself including the managed system and/or the adaptation engine.

ii. *External Uncertainty*: This type of uncertainty is a consequence of the environment that encompasses the self-adaptive software system.

I see both of external and internal uncertainty related to each other as the external uncertainty contributes in raising the level of internal uncertainty. This is because when the system reads some parameters from the external environment that holds uncertain readings, this uncertainty will be transferred to the internal model, e.g. the adaptation engine hence the internal uncertainty will grow. For example, if a robot system has some sensors for detecting obstacles in the surrounding area then this attribute, obstacle existence, represents an external source of uncertainty because the robot may fail to provide accurate readings to indicate if *certainly* there is an obstacle or not in the environment. This external uncertainty is reflected on the robot and contributes in an internal uncertainty.

2.3 Case-based Reasoning

In this section I will talk about case-based reasoning (CBR) and provide enough details for the reader about it. Section 2.3.1 provides a quick overview about CBR. Section 2.3.2 gives detailed information about CBR life cycle.

2.3.1 CBR overview

Case-Based Reasoning (CBR) is a branch of artificial intelligence. CBR can be seen as a machine learning approach [38, 37] that is build on the idea of human way of solving problems i.e. as similar problems have similar solutions [5]. CBR is established on strong mathematical foundations [31] like similarity measures which is the backbone of CBR. Any CBR system has a knowledge base that contains cases representing the knowledge of the modelled system. Each case is a pair of a problem and a solution. For any new problem, CBR systems retrieve the relevant cases from the knowledge base which contains cases, then reuse and adapt them for application on the new problem. If the adaptation and reuse phase produces new cases then they are retained in the knowledge base for later reuse. This process represents the learning mechanism, as learning is performed in CBR via retaining new cases. In the following section we will elucidate CBR working mechanism and its fundamentals.

2.3.2 CBR life cycle (RE)[4]

Figure 2.2 shows the cycle of CBR system. If we have a new problem, it has to be represented as a case, then the generating of the solution has four steps [5] as I will explain in the following:

i. *Retrieve*: The CBR system retrieves the most similar case or cases from the knowledge base by applying the similarity measures. It is a design decision to retrieve only the most similar case or a set of similar cases.

ii. *Reuse*: In this stage, the system makes use of the information of the retrieved cases. The retrieved case in ideal situation represents a solution for the problem without any modification of its information. If not, CBR adapts this information to the query problem and then formulates a new solution.

iii. *Revise*: A revision of the new solution is important to make sure that it satisfies the requirement of the system. Revising process can be done by applying it to

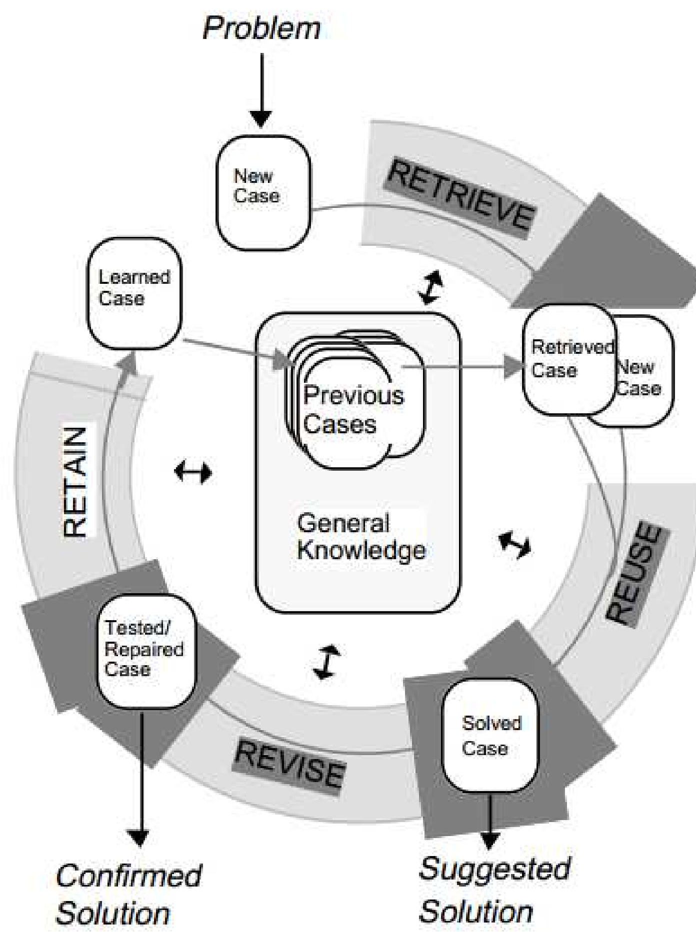


Figure 2.2: Case-based Reasoning Life Cycle [5]

real world [5] or evaluating it by domain expert. Also revising can be done by simulation approaches [38].

iv. Retain: In this stage if the new generated case represents a valuable improvement to the knowledge base, then it is saved in the knowledge base in order to use it latter.

Case can be represented in many ways such that: (1) attribute-value based representation (2) object oriented representation and (3) other specific representation like XML format. The first type is widely used because it is easy to represent the problem beside the efficiency of similarity measures used for this type. The attributes can be numeric, symbolic, data time... etc. In this thesis I will utilize the attribute-value based representation as I will explain in details in Chapter 4.

2.3.3 Similarity measures

In order to perform the retrieving process, efficient mathematical similarity measures are essential. The similarity measures are applied to the attributes of the case. The appropriateness and effectiveness of similarity measures plays an important role in the efficiency of the CBR system since efficient similarity measures will lead to a better case retrieval. This is because case retrieval is the basis for reuse and retain steps later on. Similarity measures can also be used to estimate the transferability of retained cases into new solution in the adaptation phase. Formally, a similarity measure is a function *sim: (Q,D) → [0, 1]* [37]. A value of one represent heights similarity, an exact match, and a value of zero represent the highest dissimilarity. To implement this, many traditional similarity measures exist. In [38], many similarity measures for improved case retrieval have been introduced.

Hamming Distance for example, is one of these measures for binary attributes so for any two cases x and y with n attribute vectors, the distance between these two cases can be calculated by:

$$H(x,y) = n - \sum_{i=1}^{n} |x_i - y_i| \qquad (HammingDistance) \tag{2.1}$$

Another similarity measure is *Simple Matching Coefficient SMC*:

$$sim_H(x,y) = 1 - \frac{H(x,y)}{n} |\{i|x_i = y_i\}| \qquad (SMC) \tag{2.2}$$

For numeric attributes, *Euclidean distance* can be used:

$$dist_{Euclid}(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \qquad (EuclideanDistance) \qquad (2.3)$$

Also many similarity measures like *City Block Metric* and *Euclidean Distance* can be used to calculate similarity measures efficiently. Attributes can also be weighted for improving the results of similarity measures; Weighted Minkowski Norm [38] is one of these measures. Another way for implementing similarity measures is Probabilistic Similarity Measures (PSM) [7, 19]. These similarity measures play a vital role in the proposed idea in this thesis; this will be discussed later in Chapter 4 where the uncertainty will be represented in terms of similarity between system's attributes. In order to calculate the similarity between two cases, we need to find the similarity between the corresponding attributes. This similarity is calculated by one of the similarity measures aforementioned. One important aspect in representing the case attributes is the weights of these attributes. Attributes can be weighted according to their importance. Similarity can be seen as *Local Similarity* and *Global Similarity*. The local similarity is the similarity between attributes while the global similarity is the similarity on the case level itself where attributes weights are considered.

2.3.4 Case retrieval

Retrieval process is based on similarity measures. The retrieval process may be (1) with most similar case (2) the k most similar cases or (3) all cases with minimal similarity sim_{min} . Many approaches exist for realizing retrieval process in CBR systems:

i. *Sequential Retrieval* is one way to do retrieval where the CBR system calculates the similarity for all cases first, and then the retrieval process takes place. Obviously, this approach has a drawback when the number of the cases in the knowledge base is relatively large.

ii. *2-step retrieval* suggests making the similarity calculations on a candidate subset of cases. Determining this subset is a challenge in this type.

iii. *Index-based retrieval* requires off-line generating of indices for all cases before performing similarity measures. *KD-Trees* [42] and *Fish and Shrink* [36] algorithms are used in this type.

Diversity of the retrieved cases is an issue of awareness. Suppose we have a retrieved cases $C_K = \{C_1, C_2, ., C_k\}$ for the query case C_x , a compromise between the difference of similarities among cases in C_K is vital. On the one hand and for better retrieval process, a maximal similarity between each case in C_K and C_x should be guaranteed. This will preserve the effectiveness of retrieval process. On the other hand, keeping minimal similarity among the retrieved cases in C_K will provide more alternatives for the user. To attain this diversity without affecting the quality of C_K, *Similarity Layers* [24] can be used for classifying the similarities into levels where cases with close similarities falls into one layer. Then the final retrieved cases can be chosen out of cases categorized in these layers.

2.3.5 Case adaptation

In this process, adaptation is performed on the most similar case(s) in order to provide the solution for the query case C_x. In the ideal situations; the adaptation process is limited to reusing without any modifications of the solution(s) of retrieved case(s). There are four types of case adaptation [43]:

i. Null Adaptation: In this type, the adaptation process is restricted to use the exact solution from the most similar cases without any modification.

ii. Transformational Adaptation: In this type, changes to the solution features or solution structure can be done. For example deleting, adding or modifying some part is essential to provide the new solution. The adaptation process may be limited to changes in the attributes value without changing the structure of case itself. This type will be utilized in the proposed solution in this thesis.

iii. Generative Adaptation: In this type, the solution is generated from scratch. This type of adaptation requires some heuristics to provide efficient adaptation process [30].

iv. Compositional Adaptation: In this type, the generated solution is combined from more than one solution.

2.3.6 Learning in CBR

Learning in CBR is performed by simply retaining new solutions. However; it is obvious that not all new cases should be saved. This is because the limitations of the

knowledge base size and the quality of cases stored in the knowledge base. For example it is obvious that we need not to save the solution generated by the null adaptation process, because the provided solutions are the same as those in the knowledge base, so there is no need to add them. Also, we have to be aware of retaining cases with high similarity with those retained in the knowledge base. This is because the knowledge base will be filled with cases that are highly similar to each other. Learning is not restricted to saving new cases, it also require deleting cases from knowledge base. Maintaining a high quality knowledge base requires deleting process for redundant and obsolete cases. Also noisy cases that are that were incorrectly retained should be eliminated and removed [6]. Case-Based Learning (CBL) algorithms and approaches have been covered by [6] where the learning and memorizing is carried out in CBR.

2.4 Summary

This chapter presented the background information required for the reader to follow this thesis. This chapter paved the way to read the rest of this thesis. The definition of self-adaptive software system was presented along with the classification of it. Then presented in this chapter the uncertainty principle in software engineering and in particular in self-adaptive software systems. This chapter ended with describing case-based reasoning (CBR), which is the core of the idea solution of this thesis. CBR was described in details including its life cycle, similarity measures, case retrieval, case adaptation and learning process.

Chapter 3

State of The Art

This chapter presents the related work in the area of self-adaptive software system. The process of scanning the related work to this thesis is not easy in area of self-adaptive software systems for many reasons like:

- The results sometimes were to generic to relate them to the core topic of this thesis as the main concern was the adaptation engine in self-adaptive software systems.
- The terms used in literature are overlapped and there are many expressions that refers to the same thing. For example, the term adaptation engine can be substituted with many terms like control unit [14] and autonomic manager [8]. The same issue appears in the alternative terms of "self-adaptive" for example the term "Autonomic systems" is overlapped with the term "self-adaptive" [12].
- Self-Adaptivity is contextual based and not restricted to software engineering area. Some fields like (1) artificial intelligence which concentrates in areas like games and robotics and (2) information systems which concentrates in areas like service oriented architectures and middlewares uses the self-adaptivity property.
- Self-Adaptivity has many contexts in the software engineering field itself. All software engineering activities are involved in realizing self-adaptivity starting form requirement elicitation and ending with software maintenance.

This leads to a considerable effort in deciding and intercepting the related work to this thesis.

3.1 Related work selection criteria

Some criteria were used in the search process in order to include the work in the related work. These criteria includes:

- Three libraries were searched for the related work to the self-adaptive software system adaptation engine. The libraries are (1) ACM (2) IEEE and (3) SpringerLink.
- The modularity and the maturity of the self-adaptive software system construction in the related work. Adaptation engine has to be included in the work either explicitly or implicitly as the idea of this thesis is to build an external adaptation engine.
- All the selected related work were in English language.

In the following section I will provide a list of related work.

3.2 Related work

During the last decade, the body of literature in the area of self-adaptivity has provided many of frameworks, approaches and technologies that enhance self-adaptivity. This work is widespread in many solution areas. In the following sections I will present the related work categorized according to the mechanisms used to support self-adaptivity.

3.2.1 Learning based adaptation

Salehie and Tahvildari [33] proposed a framework for realizing the deciding process performed by external adaptation engine. They use knowledge base to capture managed system's information namely domain info, goals and utility info. This information is used in the decision-making algorithm, as they name it, which is responsible for providing the adaptation decision. This framework is theoretical one which needs an implementation and evaluation which is not provided by their work.

In [22], Kim and Park provided a reinforcement learning-based approach for architecture-based self-managed software using both on-line and off-line learning. They used goal and scenario-based techniques for representing the requirements of

the system. They provided five phases for implementing self-management named Detection, Planning, Execution, Evaluation, and learning phases. In this work a case study was used to apply the approach which is a robot that requires adapting by learning from previous behaviours. This approach was supported with experimental evaluation for two experiments for game robot. Even though their results showed that this approach is effective for self-management software, but I did not see any relation between their work and architecture-based software as the name of the paper tells. This is why I categorized this work under learning adaptation not architecture-based adaptation.

FUSION was proposed by Elkhodary et al. [16] to solve the problem of foreseeing the changes in environment which hinders the adaptation during runtime for feature based systems using a machine learning technique. Knowledge base was used for selecting features. An experiment had been provided to evaluate the FUSION approach that showed accepted results. One limitation of FUSION is that it lacks operating under uncertainty.

In [20], Mohamed-Hedi et al. provided a self-healing approach to enhance the reliability of web services. They used aspect oriented programming and case-based reasoning to provide the adaptation mechanism. A simple experiment was used to validate their approach without empirical evidence.

3.2.2 Architecture and model based adaptation

RAINBOW [18] is a well-known contribution in the area of self-adaptation based on architectural infrastructures reuse. RAINBOW monitors the managed system using abstract architectural models to detect any constraints violation. Managed system's properties are captured by Rainbow using architectural style notations. A case study of two systems that share the same system concern with different adaptation styles was carried out to evaluate RAINBOW. The results showed an effective satisfaction with the system's time latency constraint.

GRAF (Graph-based Runtime Adaptation Framework) was proposed by Derakhshanmanesh et al. [14] for engineering self-adaptive software systems. Their approach represented an external adaptation engine because they separated the business logic and adaptation logic. The communication between the managed system and GRAF framework is carried out via interfaces. GRAF provided two types of adaptations: (1) Adaptation via Parameters and (2) Adaptation via interpretation of a behavioural model. The second type adapts the control flow of the adaptable

element of the managed system. They evaluated their approach by measuring memory utilization and execution performance. They conclude that there is an overhead in both metrics measures due to the migration from non-adaptive components into adaptive ones. This was normal because their approach reproduces a new adaptable version of the managed system which leads to this overhead.

Similar to GRAF [14], Vogel and Giese [40] assumed that adaptation can be performed in two ways, Parameter adaptation and Structural adaptation. They provided three steps to resolve structural adaptation and used a self-healing web application as an example. This approach comprised both managed system and adaptation logic with no separation which will face complexity problems if the number of components they are adapting increases. They implemented an application example which was a reconfiguration of components instances within EJB container. This approach lacked evaluation, as the authors assessed their solution by saying it is efficient in terms of development cost and runtime performance without any software measures.

Asadollahi et al. [8] presented StarMX framework for realizing self-management for Java-based applications. In their work they provided so called autonomic manager, which is an adaptation engine that encapsulates the adaptation logic. Adaptation logic was implemented by arbitrary policy-rule language. StarMX uses JMX and policy engines to enable self-management. Policies were used to represent the adaptation behaviour. This framework is restricted to Java-based application as the definition of processes is carried out by implementing certain Java interfaces in the policy manager. They evaluated their framework against some quality attribute. However, their evaluation for quality attributes was not quantified enough. For example they evaluated the performance by saying it is acceptable without providing any software measures.

Morin et al. [27] presented an architectural based approach for realizing software adaptivity using model-driven and aspect oriented techniques. The aim of this approach was to reduce the complexities of system by providing architectural adaptation based solution. This solution was a requirement for what they call Dynamically adaptive systems (DASs). They provided a model-oriented architectures and aspect models for feature designing and selection. This approach had four components: (1) Goal-based reasoning engine, (2) Aspect model weaver, (3) Configuration checker and (4) Configuration manager. This approach acts as software product line where variabilities are bounded at runtime and produces a set of configurations. They provide a case study without evaluation that generates two configuration scripts.

Khakpour et al. [21] provided PobSAM which is a model-based approach. PobSAM uses policies.to monitor, control and adapt the system behaviour and they used a LTL to check the correctness of adaptation. This work contains no experiment and evaluation for their approach.

The work in [13] provided a new formal language for representing self-adaptivity for architecture-based self-adaptation. This language was used as an extension of the RAINBOW framework [18]. This work explains the use of this new language using an adaptation selection example that incorporate some stakeholders' interests in the selection process of the provided service which represents the adaptive service.

Bontchev et al. [10] provides a software engine for adaptable process controlling and adaptable web-based delivered content. Their work reuses the functionality of the existing component in order to realize self-adaptivity in architecture-based systems. This work contains only the proposed solution and the implementation without experiment and evaluation.

3.2.3 Middleware based adaptation

In [9], a prototype for seat adaptation was provided. This prototype uses a middleware to support adaptive behaviour. This approach was restricted to the seat adaptation which is controlled by a software system. Their design had three layers: (1) Seat adaptation manager which is similar to adaptation engine, (2) Middleware layer and (3) Tangible layer which is responsible for sensing changes in the seat. The adaptation rules of this prototype are static and they are formatted as if-else rules. This work was not evaluated.

Adapta framework [35] was presented as a middleware that enabled self-adaptivity for components in distributed applications. They separated the business code and the adaptation logic which is considered as external adaptation engine. The monitoring service in Adapta framework monitored both hardware and software changes with two monitoring concepts: (1) Resources like CPU, memory and applications and (2) properties like CPU load usage, amount of memory and amount of application thread. I found the separation between these two concepts is not so effective as we cannot monitor a resource without reading its properties and attributes. This work lacked both experimentation and evaluation.

3.2.4 Fuzzy control based adaptation

Yang et al. [45] proposed a fuzzy-based self-adaptive software framework. The framework has three layers: (1) Adaptation logic layer, (2) Adaptable system layer, which is the managed system and (3) Software Bus. The adaptation logic layer represents the adaptation engine that includes the fuzzy rule-base, fuzzification and de-fuzzification components. This framework has a set of design steps in order to implement the adaptation. The authors did not provide any evaluation measures and contented by claiming that the framework realize the self-adaptation of software.

POISED [17] introduced a probabilistic approach for handling uncertainty in self-adaptive software systems by providing positive and negative impacts of uncertainty. An evaluation experiment had been applied which showed that POISED provided an accepted adaptation decision under uncertainty. The limitations of this approach are that it handles only internal uncertainty. Also this approach does not memorize and utilize previous adaptation decisions.

3.2.5 Programming framework based adaptation

Narebdra et al. [28] proposed programming model and run time architecture for implementing adaptive service oriented. It was done via a middleware that solves the problem of static binding of services. The adaptation space in this work is limited to three situations that requires adaptation of services. This work is supported by realistic evaluation for health care scenario.

MOSES approach was proposed in the work [11] to provide self-adaptivity for SOA systems. The authors used linear programming problem for formulating and solving the adaptivity problem as a model-based framework. MOSES aimed to improve the QoS for SOA and the work in [11] provides a numerical experiment to test their approach.

The work in [44] provided an implementation of architecture-based self-adaptive software using aspect oriented programming. They used a web-based system as an experiment to test their implementation. The used case study employed four self-adaptation scenarios with corresponding adaptation policy. Their experiment showed that the response time of the self-adaptive implementation is better than the original implementation without self-adaptivity mechanism.

Liu and Parashar [23] provided the Accord which is a programming framework that facilitates realizing self-adaptivity in self-managed applications. The usage of

this framework was illustrated using forest fire management application. In their experiment they evaluate the programming overhead of using Accord.

3.3 Discussion

After investigating the previous related work, some issues needed to be clarified with relation to CRATER.

- Many of existing work including those listed in the related work do not provide quality evaluation metrics.
- Most of the related work do not provide information regarding the limitation of their approaches and the applicable domains.
- Some of these approaches do not consider the space of the adaptation. This appears in three types:

  i. Simplistic static adaptation rules that are hard-coded [9]. It is clear that this type of solutions is not sufficient if we have large number of possible adaptations and also the maintainability cost of the system will increase. This is because each time we have a new adaptation we have to put it in the code manually and redeploy the application.

  ii. Large number of adaptations that are considered in each adaptation process [45, 17] which affects the performance negatively. I suppose that for any adaptation request we need not to search through the whole number of possible adaptation responses.

  iii. Previous adaptations are not considered in most approaches in the previous related work. This means that the system will do the same logic of adaptation many times which is a redundant computation [17].

- Some authors claim that their approaches and frameworks implement the control-loop processes. However, I found that most of them do not provide evidences regarding the design and implementation of these processes. An example of this is the work in [33].
- Self-adaptivity requires a solution outside the software engineering to provide effective adaptation. These areas of solutions are basically *Artificial intelligence*,

Fuzzy implementations and *Probability theory*. As a result, new problems may emerge in the provided solution. Suppose that we use a machine learning to solve self-adaptivity. This will lead to problems like accuracy of learning [16] for example. These problems are not related to self-adaptivity but they are technology specific consequences. This kind of problems contribute in increasing the complexity of the system which is not required.

- Uncertainty is a challenge in self-adaptive software systems which was not covered by most of the work.
- Most of work do not incorporate the knowledge component proposed by the [1]. This component should be consulted for all phases of adaptation process.

Table 3.1 summarizes the related work done in this thesis. The table has two aspects of comparison (1) Research aspects and (2) Self-adaptivity aspect. The earlier aspect is important and represent an indication regarding the maturity and creditability of the research. The later aspect is related to the topic of this thesis.

3.4 Problem statement

Based on the state of the art described in Section 3.2 and on the summary of the related work depicted in Table 3.1, it is the time to state the problem this thesis is tackling. It is obvious that handling the challenges explained earlier in Chapter 1 and investigated in Section 3.2 of this chapter is essential in order to provide an efficient self-adaptive software system as they affect the functionality, performance and trustiness of the self-adaptive software system. The majority of existing solutions fails to handle those challenges together which forms the motivation of this thesis. Based on that, the problem treated in this thesis is concluded as follow:
There is no self-adaptive software system solution that solves the following problems together :

1. *Handling and diminishing the uncertainty that hinders the adaptation process.*
2. *Managing the complexity of adaptation space by remembering the previously performed adaptations which has positive impacts on the performance of the adaptation process.*
3. *Providing an efficient performance of the adaptation engine.*

Table 3.1: Summary of Related Work

Covered literature categorization	Work	Research aspects						Self-adaptivity aspects					
		Explicit Problem. Stat.	Explicit contribution stat.	Experiment	Evaluation metrics	Limitations	Threats to validity	Adaptation Expediency, (usefulness)	Adaptation remem-brance	Uncertainty Handling	Adap. Response Time	Adap. style	Adap. engine
Learning based adapt.	[33]	✓	✓	X	X	X	X	X	✓	X	X	Dynamic	External
	[22]	✓	✓	✓	X	X	X	✓	X	X	X	Dynamic	External
	[16]	✓	✓	✓	✓	✓	X	✓	✓	X	✓	Dynamic	External
	[20]	X	X	✓	X	X	X	X	X	X	X	Dynamic	External
Architecture & model based adapt.	[18]	✓	✓	✓	✓	X	X	X	X	✓in [12]	✓	Dynamic	External
	[14]	✓	✓	✓	✓	X	✓	X	X	X	X	Dynamic	External
	[40]	✓	✓	✓	X	X	X	X	X	X	X	Static	Internal
	[8]	X	X	✓	X	X	X	✓	X	X	X	Dynamic	External
	[27]	X	X	✓	✓	X	X	✓	X	X	✓	Dynamic	External
	[21]	✓	✓	X	X	X	X	X	X	X	X	Dynamic	Internal
	[13]	✓	✓	✓	X	X	X	X	X	X	X	Static	External
	[10]	✓	✓	X	X	X	X	X	✓	X	X	Dynamic	External
Middleware based adapt.	[9]	✓	✓	✓	X	X	X	✓	X	X	X	Static	Internal
	[35]	✓	✓	X	X	X	X	X	X	X	X	Dynamic	External
Fuzzy control based adapt.	[45]	✓	✓	X	X	X	X	X	X	X	X	Dynamic	External
	[17]	✓	✓	✓	✓	X	X	✓	X	✓	✓	Dynamic	Internal
Programming framework based adapt.	[28]	X	X	✓	✓	X	X	X	X	X	X	Dynamic	External
	[11]	✓	✓	✓	X	X	X	✓	X	X	X	Dynamic	External
	[44]	✓	✓	✓	✓	X	X	✓	X	X	✓	Dynamic	Internal
	[23]	✓	✓	✓	X	X	X	✓	X	X	✓	Dynamic	Internal

3.5 Summary

This chapter covered the related work to this thesis. Related work was categorized into five categories according to mechanism the self-adaptivity is dealt with in the related work. A discussion on the covered state of the art was presented in this chapter after presenting the state of the art. Problem statement was defined in this chapter after the discussing the state of the art.

Chapter 4

Proposed Solution

Based on the research method presented in Section 1.2, this chapter contains the solution proposed by this thesis. Presented in this chapter the contribution and the explanation of how CBR is utilized as an external adaptation engine for self-adaptive software systems. Section 4.1 has an example that will be used over this chapter for clarification issue. Section 4.2 contains details about the knowledge base of CRATER and Section 4.3 provides the types of managed system attributes. Section 4.4 is the core section in this chapter that contains subsections discussing how CRATER is structured. Section 4.5 describes how utility function is used and estimated in CRATER. This chapter ends with Section 4.6 that provides information about uncertainty diminution and how CRATER deals with it.

4.1 Motivating example

This section describes a motivating example used for both explaining the solution idea of this thesis and for the validation and experimentation of CRATER. In the following subsections I will describe the a robot that needed to be self-adaptive along with its requirements and utility function realization.

4.1.1 Robot system

CRATER is utilized to receive an adaptation requests and to provide adaptation responses to be applied on managed system. In order to validate and test CRATER, I chose a robot as a managed system that demands a self-adaptive behaviour during runtime. The idea of the robot is derived from [17] with attribute extension for

more realism and variety. Figure 4.1 shows an abstract view for the used robot managed system. The robot's main task is exploratory as the robot should transmit the captured live video to a remote controlling centre. The components in Figure 4.1

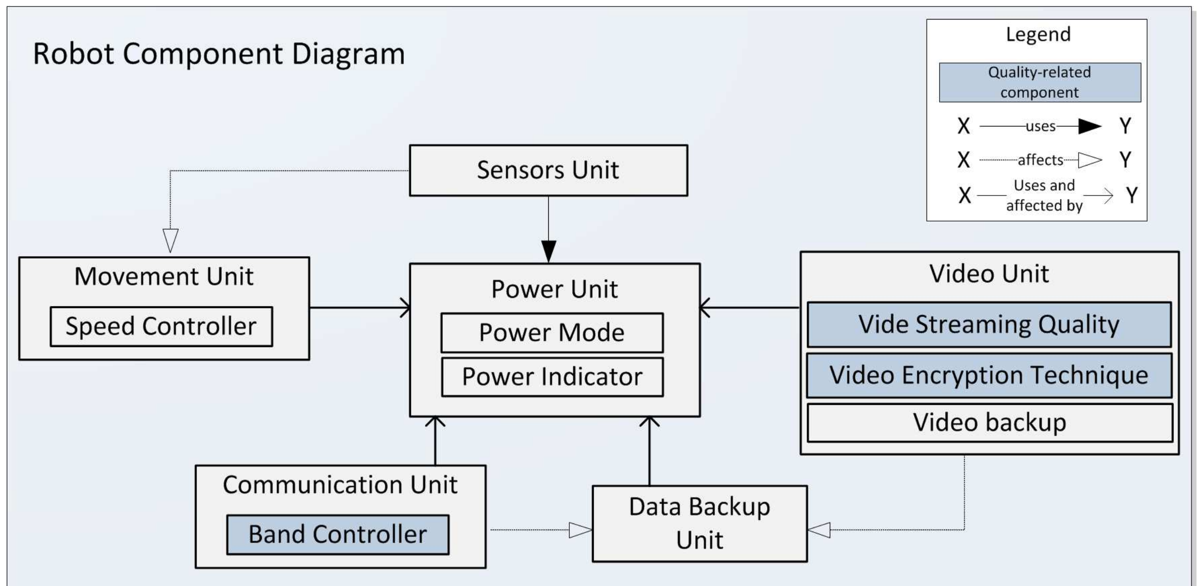


Figure 4.1: Abstract View of Robot Components

are interrelated and one component may affect the another. This interrelation contributes in having a set of possible states of the robot which is useful in explaining how CRATER works. The components in the robots are:

- *Power Unit:* which is responsible for robot's power management and supply power to the other components of the robot. I consider only two attributes of power unit namely (1) Power mode which is current operating mode like power saving mode and (2) Power indicator which represents the available remaining power.

- *Movement Unit:* which is responsible for the movement of the robot. I consider only the speed component of this unit.

- *Sensors Unit:* which represents the sensing mechanism of the robot. I consider only sensors that indicate whether there is an obstacle looming and preventing robot during runtime, like heat and objecting mass, or not.

- *Video Unit:* which is responsible for video streaming functionality of the robot. I consider the attributes video quality and video encryption.
- *Data Backup:* which is responsible for storing video in case of the communication unit fails to work.
- *Communication Unit:* which is responsible for the communication with the remote centre. I assume that communication unit is used for live transmitting the of video that the robot is capturing and works within some communication bands such VHF and UHF.

Table 4.1: Robot attribute data sheet

Attribute	**Values**
Communication	$\{OFF, VHF, X - band, UHF\}$
Power Mode	$\{Full\ Power, Medium\ Power, Saving\ Mode\}$
Power Indicator	$\{Low, Medium, High\}$
Speed	$\{Low, Medium, High\}$
Video quality	$\{Very\ low, Low, Medium, High, Very\ high\}$
Data Backup	$\{On, Off\}$
Obstacles	$\{On, Off\}$
Encryption	{*Zig-Zag Permutation, Puer Permutation, Naive, Video Encryption Algorithm (VEA)* }

Figure 4.1 shows some quality-related components. These components has a direct impact on the quality of delivered service by the robot. For instance the component "Video Encryption Technique" affects the security of the transmitted data. Similarly the component "Communication Band" affects quality of communication channel. Table 4.1 shows the robot attributes set with their values. Un-adaptable attributes are Power Indicator and Obstacles which represent a read only attribute and can not be altered during the adaptation process. The rest are adaptable attributes and their values can be changed during the adaptation process.

4.1.2 Robot goals

Generally, managed system's goals are the functioning and quality requirement provided by the customer and related to adaptation process. The following subsections are dedicated to explain the system goals of the robot.

4.1.2.1 Quality-related requirements

Table 4.2 shows the robot's quality-related goals. Trade-off between the quality attributes is required to provide a better overall quality of the robot. For example the security attribute presented in Table 4.2 requires compromising when selecting one of them because they are affected by the power unit. Table 4.3 shows all of encryption techniques along with their characteristics. Robot task is to choose one among them according to robot's state. This is because high security encryption techniques requires more computational effort which requires more power consumption. This means that the robot should choose the encryption technique that suits its state, more precisely its power state.

Table 4.2: Robot quality requirements

Goal	**Descriptions**
Transmission Security	This goal is about keeping the transmitted data as secure as possible. This is done by selecting one among set of encryption algorithm as each of them has its advantages and drawbacks. Table 4.3 shows example of trade-offs among these algorithms.
Video Quality	This goal is about keeping the video quality as better as possible. This is done by selecting the appropriate video quality during runtime. Power affects this goal as higher qualities requires more power consumption.
Communication quality	This goal is about keeping the communication channel as fit as possible. Some communication bands requires more power and some of them has low coverage.

Table 4.3: Encryption techniques characteristics

Technique	Security level	Encryption performance
Zig-Zag Permutation	Very low	Very fast
Puer Permutation	Low	Super fast
Naive	High	Slow
Video Encryption Algorithm (VEA)	High	Fast

4.1.2.2 Functioning requirements

In addition to the quality requirements, a set of functioning requirements are essential for robot operation. Table 4.4 shows the functioning requirements that keeps the robot fit and provides the required functions in terms of adaptation process.

Table 4.4: Robot functioning requirements

Goal	Descriptions
Power consumption	The robot should change its power mode according to power indicator reading. If power indicator reading is 'low' and the power mode is 'Full power' for example, the robot should alter the power mode and reduce it e.g. 'Saving mode' and reflects this change to other components.
Robot Fitness	The robot should maintain its fitness and manage the relation between the speed and power. The robot should reduce its speed if (1) the power is not sufficient or (2) an obstacle is detected i.e. 'Obstacle' attribute has a value 'true' . The same thing is applied on the relation between power mode and video quality as higher video qualities requires more power.
Data backup	If the communication with the remote centre is lost, the robot should enable the data backup till the communication is on again. This requires to reduce the video quality due to limitation on the space of backup storage.

4.2 CBR knowledge base

A knowledge base intuitively saves the cases for future retrieval process. In CRATER the knowledge base saves the states of the managed system such that no case in the knowledge base contains goal violations of the managed system. The correctness of the knowledge base i.e. the knowledge base that contains only desirable states of the managed system, is guaranteed in the retain process where no case is retained unless it has a utility greater than UT [1]. Knowledge base is modelled with domain experts by capturing all managed system's attributes that are related to the adaptation process. The operation performed on the knowledge base is restricted to (1) case retrieval and (2) case retain. An advantage of the knowledge base is that the domain expert can investigate it for offline maintenance i.e. add new cases, remove cases and alter cases. The cases stored in the knowledge base always have a utility value greater than UT. This facilitates the retrieval process and provides always an efficient adaptation response that saves the system from running in unwanted sates. The quality of cases stored in the knowledge base can be controlled by retaining only cases which have utility greater than certain value. This option is vital in keeping the knowledge base effective and efficient. Based on the example in section 4.1 the knowledge base contains a set of cases that holds the values of robot attributes operating in desired states which means they have utility greater than UT.

4.3 Managed system attributes

Managed system operating states and configurations are modelled as CBR cases. Each case has a set of attributes that have both types and weights as will be explained in the following subsections.

4.3.1 Attribute types

Case attributes can be flagged as one or more of the following types in Table 4.5 assuming that no attribute can take two contradicting types e.g. Adaptable and Unadaptable at the same time. *Utility threshold breaker* attribute type is used during the analysis process of the managed system state because the adaptation process will alter the values of these attributes. *Utility antagonist* attribute type is used to

[1]see Section 1.3 for the definition of Utility Threshold

indicate the attributes that contributes in reducing the utility of the managed system. Utility antagonist can be used in CRATER to provide the best adaptation response.

Table 4.5: Managed System Attribute Types

Attribute Type	Description
Adaptable	Denotes an attribute whose value can be changed during the adaptation process like *Speed* attribute.
Un-adaptable	Denotes an attribute whose value can not be changed during the adaptation process like *Obstacles* attribute.
Utility threshold breaker	Denotes an attributes whose value contributes in providing goal violating state.
Utility antagonist	Denotes an attribute whose value contributes in decreasing the utility of the managed system.

4.3.2 Attribute weight

It is normal that attributes do not have the same effect on the managed system state. Some of the managed system's attributes have greater effect than the others. Based on that, Pareto principle [29] is considered and each attribute is weighted in order to provide optimal representation and modelling for the state of the managed system. In addition, attributes weighting is an essential process for two reasons:

- If the adaptation request has un-adaptable attribute among the UT breaker attributes then we can only change the adaptable attribute values. CRATER knows how to do that my means of weighting. All un-adaptable attributes must have values grater than the adaptable attributes values. This directive modelling helps CRATER to provide a meaningful and applicable adaptation response for the managed system.

- Weighting is essential also in quantifying and measuring uncertainty in adaptation request.

4.4 CBR as adaptation engine

The main idea of this thesis is using case-based reasoning (CBR) as an external adaptation engine. This idea is inspired form the compatibility between CBR life cycle discussed in 2.3.2 and shown in Figure 2.2 from one side and the closed control loop and adaptation process discussed in 2.1 and shown in Figure 2.1. More precisely the CBR engine works as the process *Deciding* in Figure 2.1 which is an important process in the self-adaptation process that provides the adaptation actions. CBR is used as external adaptation engine that embraces the knowledge base of the self-adaptivity related attributes of the managed system. This external adaptation engine receives adaptation request and responds with adaptation response as will be described in the next sections. The novelty of CRATER is that it utilizes not only the similarity measures but also the utility functions in both the assessment of the retrieved cases from the knowledge base and in constructing new adaptations. The use of utility functions enables CRATER to startup from empty knowledge base which represents a challenge in the applications of CBR.

Figure 4.2 shows the mechanism of CRATER and how it works. In the following subsections I will present detailed information about each the components in that figure.

4.4.1 Adaptation request

As explained in Section 1.3, the adaptation request is a managed system's state that violates the managed system's goals. Adaptation requests are sent in reality when the system enters or reaches unwanted state which should be overcame by the system. Adaptation requests in CRATER contain the attributes of the managed system at the point of the violation. This means that CRATER treats self-adaptivity in a reactive or proactive way depending on the implementation of monitoring process. Concisely, adaptation request A_{req} is a set of attributes $\{A_1, A_2, ..., A_n\}$ that represents the managed system at adaptation initiation point. A subset of these attributes are adaptable attributes. Adaptation request in CRATER is any state that breaks the UT [2] of the managed system. Adaptation request is sent to the adaptation engine, which is the CBR engine, in order to do the adaptation process on it. So as to provide adaptation request, a monitoring process is needed as shown in Figure 2.1. The task of the component *observe and decide* is to monitor the managed system and detect

[2]section 4.5 provides details about system utility

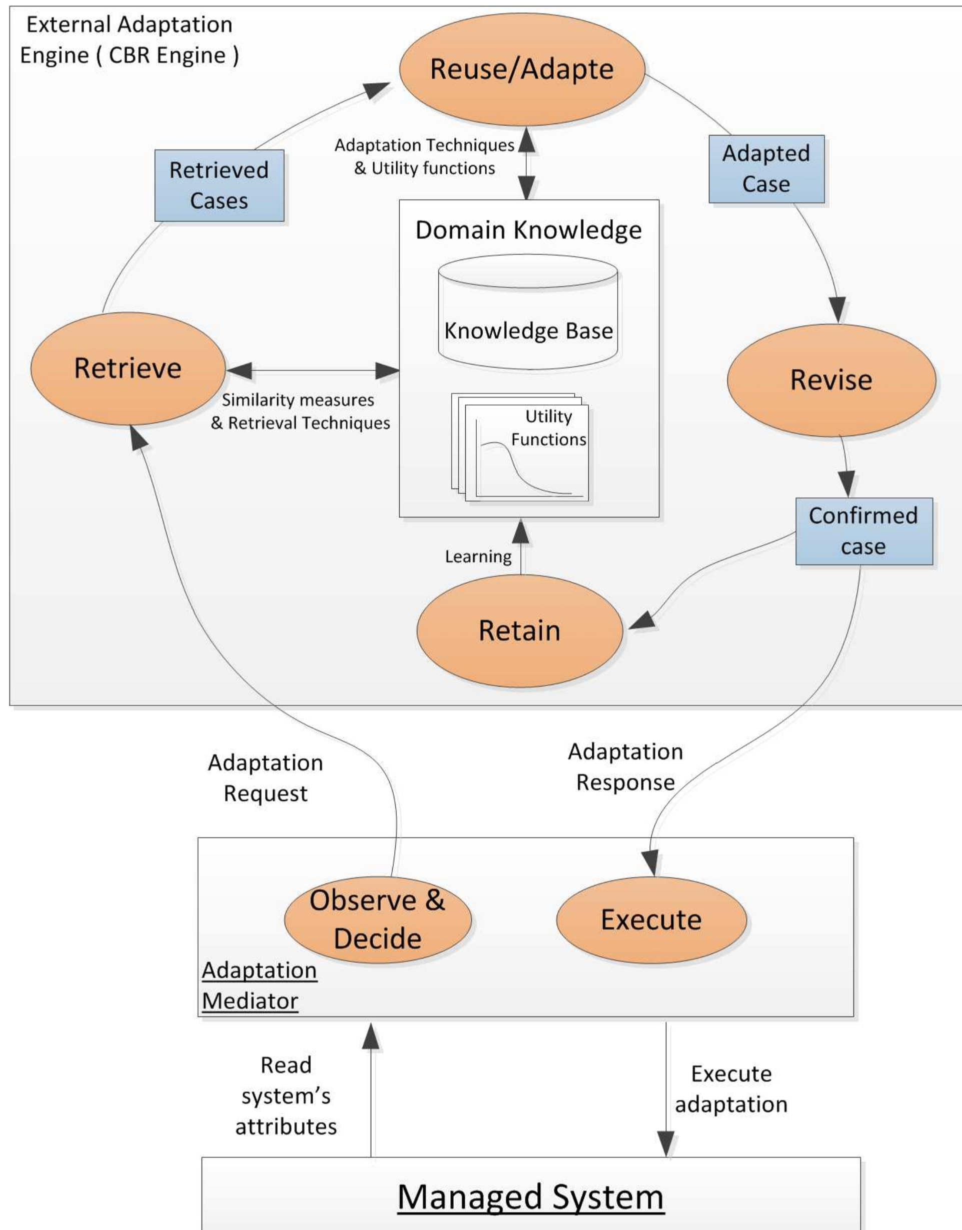


Figure 4.2: CRATER Reference Model

if there is a violation in the state of the managed system. If the monitor detects a violation then an adaptation request is formed by collecting managed system's attributes' values and pass them to the CBR engine as *adaptation request.* An example of adaptation request is {*Power* ↦ *Full power, Video Streaming quality* ↦ *Very High, Obstacles* ↦ *True, Speed* ↦ *Fast*}. This is a robot state that contains a set of values for a robot system's attribute. This state represents unwanted operating state assuming that the robot speed should not be fast if an obstacle is detected. This state is a typical adaptation request because of goal deviation which is a utility value breaking the utility threshold.

4.4.2 Adaptation response

Adaptation response is the result of the adaptation process generated by the CBR engine. It is an object that contains a corrective operating state that rescue the managed system from its violating state. Adaptation response must have a utility greater than UT value. Obviously the adaptation response with greater utility value is better. To apply the adaptation response, an *execute* component is mandatory. If the adaptation response succeed in overcome the UT of the managed system, this adaptation response is retained in the knowledge base for future use. An adaptation response for the adaptation request in the previous section is {*Power* ↦ *Full power, Video Streaming quality* ↦ *Very High, Obstacles* ↦ *True, Speed* ↦ *Slow*}. It is obvious that the cause of unwanted state is the high speed while detecting an obstacle. The corrective adaptation response changes the speed to *Slow* which raises the utility of the robot and make it greater than UT.

4.4.3 Adaptation process

Adaptation process is the process that starts when the CBR engine receives adaptation request and ends with providing adaptation response. Algorithm 1 summarizes this process. The following subsections describes in details the adaptation process.

4.4.3.1 Analysing adaptation request

When the CBR adaptation engine receives an adaptation request it analyses it to identify attributes that causes UT break and attributes that abate the managed system utility. This identification helps in providing efficient adaptation response by

Algorithm 1 CRATER adaptation process

Require: *KB* , A_{req}
Ensure: $Utility(A_{res}) > UT$

```
1: List cases ⇐ Retrieve (KB,A_req)
2: List qualifiedAdaptationFrame
3: Case A_res
4: while Case c ⇐ Iterate(cases) do
5:     if Sim(A_req,c) ∈ [1,β] then
6:         qualifiedAdaptationFrame.add(c)
7:     end if
8: end while
9: if qualifiedAdaptationFrame is not Empty then
10:    A_res ⇐ max(CaseExpediency(qualifiedAdaptationFrame) )
11:    Return A_res
12: else
13:    A_res ⇐ ConstructiveAdapt(A_req)
14:    Retain(A_res,KB)
15:    Return A_res
16: end if
```

changing the values of these two types of attributes to get higher utility as much as possible.

4.4.3.2 Case retrieval

Case retrieval is a CBR core functionality. We retrieve the most similar case or cases to the adaptation request. The adaptation request is formulated by excluding adaptation request's attributes that break UT form retrieval calculation process. This exclusion is inevitable because the knowledge base keeps only the best operating states which mean that no case in the knowledge base has attribute values that break UT. Then exclusion becomes logical because by doing it the similarity between the adaptation request and cases stored in the knowledge base becomes more realistic. An example for excluded attribute based on the example in section 4.1 is *Speed* attribute. This is because the speed causes the robot utility to break utility threshold. Even though both *Speed* and *Obstacles* contribute in breaking the UT, we exclude just speed attribute because it is adaptable attribute unlike the obstacle attribute which is not adaptable.

4.4.3.3 Constructing QAF

The retrieval process for adaptation request $Adaptation_{Req}$ returns a set of cases C_K such that each case C_k in this set satisfies the condition:

$$\beta \leq Sim(C_k, Adaptation_{Req}) \leq 1 \tag{4.1}$$

where β is a value between [0,1] that 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 request and each retrieved case. *Sim* function is implemented in the CBR engine implementation [2]. The set of cases that satisfies the previous condition are called *Qualified Adaptation Frame*. So β is the sufficient similarity for qualifying a case to the qualified adaptation frame. Integrating β has the advantages: (1) it provides an alternative options by providing more than one case in the provided qualified adaptation frame (2) exclude non-related cases to the adaptation request and (3) utilizes the knowledge in the similar cases because the similarity is not the only decisive criteria in providing the adaptation response as the utility of the case is also important. Suppose that a PC recommender system

utilizes both CBR and utility. The user of the system demands a customized PC with 3GB of RAM. The result of the retrieval returns two identical PCs except the amount of RAM. The first result has similarity 100% to the user query with 3GB of RAM and 0.95 utility and the second result has similarity 97% with 4GB of RAM and 0.98 utility. It is obvious that even the first result is what the customer wanted from the beginning, but he/she could be interested in the second PC where the amount of RAM is more which increases the utility. This is why an integration between the similarity and the utility should be considered when choosing the adaptation response case from the qualified adaptation frame. The combination between the similarity and the utility is called case expediency. If we want to consider only the identical case i.e. cases with similarity of 100% to the adaptation request then we set β to one. In this situation the qualified adaptation frame will contain only a case with 100% similarity to adaptation request if exists in the knowledge base. As shown in algorithm 1 the returned case i.e. the adaptation response is the case that satisfies the previous condition in equation 4.1 i.e. it is in the qualified adaptation frame and has the highest expediency. The expediency of a case in the qualified adaptation frame is calculated in CRATER by Equation 4.2:

$$CE(c)_{QAF} = 1 - [(1 - sim(Adap_{req}, c)) * utility(c)] \tag{4.2}$$

where CE is the *case expediency* for a case c in the qualified adaptation frame and sim is the similarity between the adaptation request $Adap_{req}$ and the case c. Equation 4.2 returns one if the similarity is one which represents the heights level of similarity and of course the utility is greater than the utility threshold as all cases in the knowledge base have utilities greater than utility threshold. If the utility is one then the case expediency value equals the similarity value. This novel combination in calculating case expediency is crucial:

- On the one hand the inclusion of similarity of the retrieved case in calculating case expediency is important as higher similarity leads to less changes in the managed system state which is an important issue. If the similarity equals one, which means that the $Adap_{req}$ and the case c are identical, then the case expediency is one.

- On the other hand the inclusion of the utility of the case is not less important than similarity inclusion that is because the utility reflects the quality of meeting managed system's goals.

Noting that all cases in the knowledge base represents a non-violating states, retrieving cases that are not utility breaker is realized by default and the adaptation process always returns a case with utility greater than UT. If β is set to 1 then the qualified adaptation response is the case that has similarity of *one* and certainly a utility greater than UT. In this case if the qualified adaptation frame is not empty, it contains only one adaptation response that is unique because the knowledge base will not save two identical cases. This is guaranteed by the nature of CRATER mechanism as CRATER do not generate new case unless the knowledge base fails to provide the required adaptation response. So only the newly generated case is saved in the knowledge base.

4.4.3.4 Generate adaptation response

If no case satisfies the previous condition e.g. the qualified adaptation frame is empty, CRATER will generate the adaptation response based on the utility function by adapting the adaptation request attributes in order to provide a case with utility greater than UT. This is done with the help of the previously identified attributes that break UT. This process is called *Utility-guided constructive adaptation* which has two flavours:

- *First Fit Heuristic*: It is a normal iterative search process in the space values of the attributes [30] applied on the adaptation request that breaks UT. The first values that cause the utility of adaptation request greater than UT is returned as adaptation response and the search stops. Algorithm 2 explains this type of adaptation.

- *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. Algorithm 3 explains this type of adaptation.

If the adaptation response is generated by one of the previous ways, the utility of the generated case is considered as the expediency.

4.4.3.5 Retain

Retain phase is restricted to the newly generated adaptation response from the Utility-guided constructive adaptation process. Because all generated adaptation response

Algorithm 2 First Fit Heuristic Constructive Adaptation

Require: A_{req} , *KB*
Ensure: *Utility*(A_{res}) > *UT*

1: *Case* A_{res}
2: *List UTbreaker* ⇐ *Analyse*(A_{req})
3: **while** (*AttributeValue* a_v ⇐ *Iterate(Values(UTbreaker))*) **do**
4: A_{res} ⇐ *apply*(A_{req} ,a_v)
5: **if** *Utility*(A_{res}) > *UT* **then** stop
6: **end if**
7: **end while**
8: **Retain** (A_{res},*KB*)
9: **Return** A_{res}

Algorithm 3 Best Fit Heuristic Constructive Adaptation

Require: A_{req} , *KB*
Ensure: *Utility*(A_{res}) > *UT*

1: *Case* A_{res}
2: utilityValue ⇐ 0
3: *List UTbreakerAttributes* ⇐ *Analyse*(A_{req})
4: **while** (*AttributeValue* a_v ⇐ *Iterate(Values(UTbreakerAttributes))*) **do**
5: *Case temp* ⇐ *apply*(A_{req} ,a_v)
6: **if** *Utility(temp)* > *utilityValue* **then**
7: *utilityValue* ⇐ *Utility(temp)*
8: A_{res} = *temp*
9: **end if**
10: **end while**
11: **Retain** (A_{res},*KB*)
12: **Return** A_{res}

from both first fit and best fit has a utility greater than UT, they are qualified for retaining in the knowledge base for future reuse. It is clear that CRATER is able start operating with empty knowledge base which considered an advantage. The utility function governs the learning process which guarantees the quality of retained cases. During the runtime of CRATER, the number of retained cases in the knowledge base will increase which raises likelihood of returning adaptation response instead of generating it from scratch. This has a positive impacts on the performance of CRATER and reduces the response time significantly. Figure 4.3 depicts the adaptation process performed by CRATER as a flow chart.

4.5 Utility function

This section provides information and details about utility function and how it is used in CRATER. The following subsections contains intensive explanation of that.

4.5.1 Utility function importance

Utility function is used in CRATER to capture system goals. I combined the utility function concept in the functionality of CRATER in order to:

- Incorporate functional and quality requirement assessment for the cases.
- Provide a heuristic for the adaptation generation process.
- Provide affirmation regarding the adaptation response usefulness and expediency.
- Analysis the adaptation requests in order to identify UT breaker attributes.
- Determine the managed system desirable and undesirable states which are crucial for the adaptation process.

4.5.2 Utility function definition

Utility functions represents the heuristics for both (1) identifying the cause of adaptation request by inspecting the managed system attribute or attributes that originate this adaptation request and (2) generating adaptation requests in CRATER. Basically

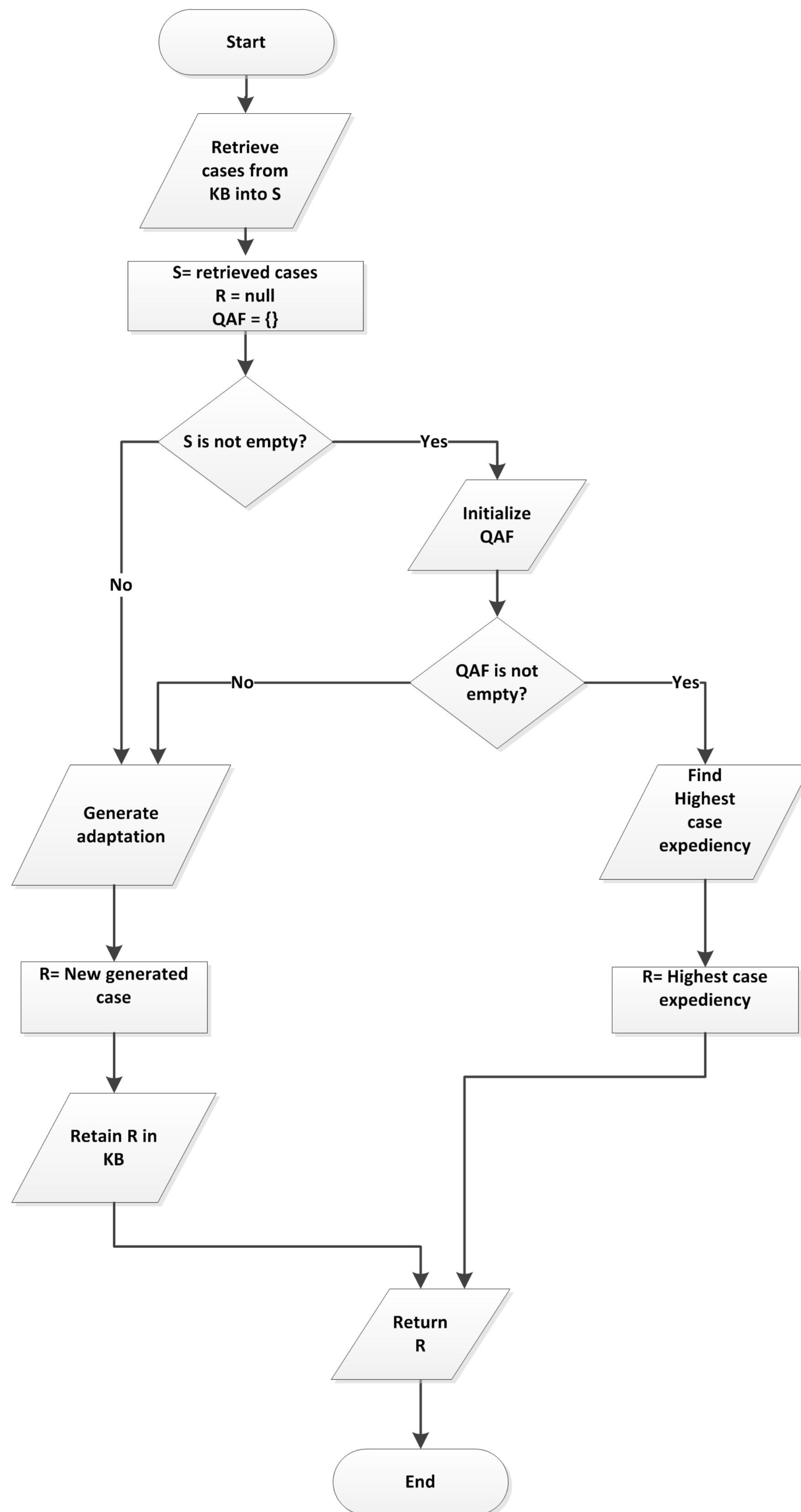


Figure 4.3: Adaptation Response Estimation Flow Chart

utility function is a function that maps a set of attributes to a value if certain condition holds. Normally the certain condition represents attributes satisfactory values. For a simplicity, the utility function definition in this thesis is elaborated based on the work in [25] and extended in order to combine multiple utility-involved attributes. Generally, the utility function in CRATER is defined in Equation 4.3 an looks like:

$$Utility_{(a_1,\ldots,a_i)} = \begin{cases} v_1 & \text{if } condition_1 \text{ holds} \\ v_2 & \text{if } condition_2 \text{ holds} \\ . & \\ . & \\ v_{n-1} & \text{if } condition_{n-1} \text{ holds} \\ v_n & Otherwise \end{cases} \tag{4.3}$$

where:

- *(a_1,...,a_i)* is the set of involved managed system attributes for this utility function.
- *(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.

4.5.3 Utility function weight

In reality the adaptation involved attributes in the managed system can be represented by more than one utility function due to the correlated effects among attribute. As a result one attribute can be involved in more than one utility function. Weighting these utility functions is a crucial issue in modelling managed system goals. Weighting process simply attached a weight to each of the utility functions. The more the weigh is the more important this utility function is. Weighting process normally is the task of the domain expert.

4.5.4 Overall utility function

If the managed system's goals has been captured as many utility functions then we need a solution for finding the overall utility of the managed system in terms of its utility function. *Weighted Geometric Mean* (WGM) is used for that end because the

WGM is affected by all utility function values and also works if one of the utility values is zero. If we have a set of utility functions 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:

$$U_{overall} = (\prod_{i=1}^{n} u_i^{w_i})^{1/(\sum_{i=1}^{n} w_i)} \tag{4.4}$$

4.5.5 Utility function examples

As explained earlier the utility function is used mainly to capture managed system goals. In this section I will present examples of utility functions used to represent robot's goals explained in Section 4.1.2.

An example of utility function looks like:

$$Utility_{(Speed,Obstacles)} = \begin{cases} 0.1 & if(\text{Speed=High and Obstacles=True })holds \\ 0.8 & if(\text{Speed=Medium and Obstacles=False})holds \\ 1 & \text{Otherwise} \end{cases} \tag{4.5}$$

In the example shown in Equation 4.5, assuming that utility threshold is 0.1, the utility value of this utility function equals 0.1 if the robot's speed is $HIGH$ and the robot detects an obstacle in the surrounding which is represented by $TRUE$ value for the attribute obstacles. Utility function values normally should have values [0,1] where 0.1 value represents a violation of system goals and *one* represents the ideal operation state. If the utility value is less than or equal UT then the state of the system at this point represents an adaptation request and the attributes involved the utility function that has UT breaker value should be changed to get rid of the cause of violation. In our example if we want to react to the adaptation request generated because of the $Utility_{(speed,obstacle)}$ utility function then it is obvious that we can not change the obstacle attribute because it is not adaptable attribute and we can only change the value of speed. Logically we have to reduce the speed of robot and make it *Medium* or *Low* depending on what value will make the higher utility. The *otherwise* condition in the example makes the speed value *Low* to provide maximum utility for this function.

$$Utility_{(PM,VQ)} = \begin{cases} 0 & if(\text{PM=1 and (VQ=5 or VQ=4) })holds \\ 0.4 & if(\text{PM=1 and VQ=3})holds \\ 0.5 & if(\text{PM=2 and (VQ=5 or VQ=4) })holds \\ 0.99 & \text{Otherwise} \end{cases} \quad (4.6)$$

The utility function described in Equation 4.6 represents the relation between *Power Mode* and *Video Quality*. If the power mode is 1 i.e. 'Low' and the video quality is 'Very High' or 'High' then the value of the utility function is zero. Similarly, if the power mode is 'Low' and the video quality is 'Medium' the value of this utility function is 0.4. Zero utility functions represents a robot state where an adaptation is required. The zero value of any utility function cause the overall utility function equals to zero because I use *Weighted Geometric Mean* for calculating the overall utility as described in Section 4.5.4. Each utility function has a weigh to represent its effect. The utility function can describe the behaviour of one attribute and also the relationship between two attributes or more. The utility function can represent the relation among n attributes. For example if we want to represent the utility function among the attributes {*Power Mode, Video Quality, Transmission Security*} then the utility function for these three attribute looks like Equation 4.7:

$$Utility_{(PM,VQ,TS)} = \begin{cases} 0.1 & if(\text{PM=3 and (VQ=1 or VQ=2) and TS=1) } holds \\ 0.5 & if(\text{PM=2 and VQ=2 and TS=1) } holds \\ 0.8 & if(\text{PM=1 and VQ=3 and TS=3) } holds \\ 0.99 & \text{Otherwise} \end{cases} \quad (4.7)$$

Thus the nature of the utility function definition in CRATER enables incorporating many attributes which provides a comprehensive way in capturing the managed system goals. The rest of the utility functions of the robot is defined in the same manner. It is normal to have more than one utility function with utility value less than or equal UT breaker value. All UT breakers should be handled and resolved in order to provide efficient adaptation response.

4.6 Uncertainty diminution in CRATER

Uncertainty is a challenge that hinders the adaptation process. In order to continue in this section I need to figure out some assumptions and intuitive issues regarding

uncertainty handling in CRATER:

- Exact utility of the managed system state is not deterministic in the presence of uncertainty. Instead all possibilities of uncertain data should be considered for providing efficient handling of uncertainty.
- In order to evaluate and handle uncertainty in the managed system state we need to measure uncertainty. This measurement is important as it provides directives for the adaptation process under uncertainty.
- Needless to emphasis that raising the utility of the managed system is an ultimate goal particularly in CRATER. If CRATER faces uncertain state of the managed system then it tries to find the best adaptation response that raises the utility of the managed system.

4.6.1 Uncertainty handling

Generaly there is two ways in dealing with uncertainty in the context of utility function:

- *Optimistic Paradigm:* Which deals with the uncertain values as values that heighten the utility.
- *Pessimistic Paradigm:* Which deals with the uncertain values as values that belittle the utility.

Even though both of these two paradigms has its own advantages and drawbacks, but they do not provide an effective way of handling uncertainty particularly if more than one attribute in the uncertainty-concern object has uncertain values. CRATER follows a *Hybrid Paradigm* where it analysis the uncertain situation for better and efficient adaptation by:

- Analysing the uncertain managed system state and construct a tree of all possible states and then calculates overall uncertainty η.
- If at least one of the elements of this tree represents adaptation request i.e. a state that breaks the UT, then CRATER issues adaptation process.
- The issuance of adaptation process under uncertainty can be controlled in CRATER via η i.e. if η is one then CRATER behaves pessimistically while

when η is zero CRATER behaves optimistically. Normally η has a value greater than zero and less than one and CRATER considers only uncertainties less than or equal this value.

According to the three dimensions of uncertainty discussed in Section 2.2 the following subsections show how uncertainty is classified, quantified and handled in CRATER.

4.6.2 CRATER's uncertainty location

Uncertainty location is the location where uncertainty appears. It is important to locate uncertainty in order to measure and diminish it. More specifically uncertainty appears within CRATER model in (1) the adaptation request attributes' values in the adaptation requests and (2) in the qualified adaptation frame. The former is due to knowledge shortage in the adaptation request attributes values. This could be due to environment reasons or managed system measurement errors in providing known values. The later is due to variability in adaptation responses in case that we have more than one adaptation response in the qualified adaptation frame with the same case expediency. This means that CRATER is uncertain regarding which case to return as adaptation response among cases in the qualified adaptation frame.

4.6.3 CRATER's uncertainty level

Uncertainty level is a quantification for uncertainty. In CRATER we need to specify the degree of uncertainty in all locations it appears in e.g. the adaptation request and in the qualified adaptation frame in order to consider and handle it later on the adaptation process.

4.6.3.1 Adaptation request uncertainty

The uncertainty in the adaptation request is a result of uncertain state of the managed system. If the managed system has uncertain values at least in one of its attributes then we need to know if this state demands an adaptation process or not. In order to perform that we need to analyse the managed system's state. Suppose one of managed system attributes has uncertain values represented by question mark '?'. The analysis process converts this state into set of states that contains all possible states by replacing the question marked values with all possible values assuming the

managed system has a predefined values space for all attributes. For example if the managed system is modelled as set of attributes $\{A_1, A_2, A_3, A_4\}$ and each attribute has a set of possible values $\{1, 2, 3\}$. Managed system state could be $\{1, 1, 2, 1\}$ which represents a state with certain values i.e. certain state. However, for some reason the state of managed system could be $\{?, 1, 3, 2\}$ which represents a state with uncertainty. This sates requires the analysis process in order to check whether this state holds an adaptation request or not and what is the probability of that. To perform this process, we need to make all possible states out of this uncertain state which is the set κ of all combinations: $\{\{1, 1, 3, 2\}, \{2, 1, 3, 2\}, \{3, 1, 3, 2\}\}$. The next step is to estimate the utility of each state in κ in order to determine the number of states $\Re$ that requires adaptation process e.g. UT breaker states. CRATERS perform adaptation for any state in κ that breaks the utility threshold and chooses the case with maximum case expediency as an adaptation response. Choosing the highest adaptation expediency among all adaptation responses is based on the assumption discussed above that the ultimate goal is to raise the utility of the managed system. After this explanation we can quantify two things: (1) the adaptation request ratio (μ) of managed system uncertain state and (2) the degree of uncertainty in the managed system state.

Adaptation request ratio (μ) is a value $\in [1, 0]$ and determined as shown in Algorithm 4 by the following equation:

$$\mu = \frac{\Re}{Size(\kappa)} \tag{4.8}$$

The uncertainty degree in the managed system state (Θ) is a value $\in [1, 0]$ and estimated by:

$$\Theta = \frac{\#uncertain\ attributes}{\#all\ state\ attributes} \tag{4.9}$$

Equation 4.9 returns zero if there is no attribute in the adaptation request with uncertain values. It returns one if all attributes of the adaptation request are uncertain values.

It is important also to estimate the overall uncertainty η in the managed system state with uncertain values. If (μ) equals one, which means that all states in the set (κ) are utility threshold breakers, this means that it is the highest level of uncertainty in the managed system state and makes (η) equals also one. The same reasoning is applied on (Θ) which means that if (Θ) is one i.e. all attributes of the managed system state

Algorithm 4 Estimating μ

Require: *UncertainState* ***S***

Ensure: ***S*** *has uncertain values*

1: *List* $\kappa \Leftarrow$ *PossibleStates(S)*
2: $\Re \Leftarrow 0$
3: **while** κ *has more elements* **do**
4: *State temp* $\Leftarrow$ *Next*(κ)
5: **if** *Utility(temp)* $\leq$ *UT* **then**
6: *increment* $\Re$
7: **end if**
8: **end while**
9: **Return** $\frac{\Re}{Size(\kappa)}$

are uncertain, and then (η) is one also. Based on this argument (η) is calculated in CRATER by the following equation:

$$\eta = 1 - [(1 - \mu) * (1 - \Theta)] \tag{4.10}$$

The overall uncertainty(η) is useful in providing insights regarding (1) uncertainty in the managed system state and (2) the appropriateness of adaptation response. The overall uncertainty (η) is a variable in CRATER model. This means that CRATER will issue adaptation process if the managed system uncertain state's (η) estimation is less than or equal a predefined value e.g. if (η) equals 0.8 in CRATER configurations and the managed system state's (η) value is greater than 0.8 then no adaptation is issued due to high percentage of uncertainty. Determining (η) is crucial issue because it directs CRATER when to issue adaptation process if the managed system state is uncertain and to avoid risky adaptations. Suppose (η) value in the managed system state is 0.99 which means that the managed system uncertainty is too high which contributes in (1) bigger variety of possible states and (2) more processing and computation done by the adaptation engine. By limiting (η) to some value e.g. 85% CRATER managed to provide adaptation response at reasonable response time.

4.6.3.2 Uncertainty in qualified adaptation frame

Uncertainty in the qualified adaptation frame occurs if there exist more than one case satisfying the highest expediency. For example if the qualified adaptation frame

has two states with *(similarity,utility)* of *(0.95,0.8)* and *(0.9,0.4)*. Based on the Equation 4.2 both cases have the same case expediency which is 0.97. This situation represents uncertainty for CRATER and the selected case is the case with the higher utility because the ultimate goal of the adaptation process is to raise the utility of the system as much as possible as assumed in Section 4.6. The uncertainty in the qualified adaptation frame is estimated in CRATER by the following equation:

$$U_{QAF} = \frac{N}{Size(QAF)} \tag{4.11}$$

where N is the number of cases with identical highest expediency.

4.7 Summary

This chapter covered in details the idea solution part of this thesis. A motivating example was presented to illustrate the need of self-adaptivity. A detailed description of the proposed solution was discussed in this chapter including how CBR is utilized as an adaptation engine and the managed system attributes modelling. The usage of utility function in CRATER was introduced in this chapter describing how it is utilized to represent the managed system's goals. Uncertainty handling in CRATER was also presented in this chapter. Uncertainty quantification, uncertainty location determination and uncertainty degree were presented and explained in the CRATER uncertainty handling mechanism.

Chapter 5

Implementation

This chapter provides details regarding the implementation of CRATER. Section 5.1 contains information about the implementation and the architecture of CRATER. Section 5.2 provides information about the generation of adaptation requests and Section 5.3 explains how the monitoring process is implemented. Section 5.4 illustrates the implemented classes while Section 5.5 shows the used tools in the development process. This chapter ends with Section 5.6 which describes how uncertainty is realized, injected and dealt with in the prototypical implementation of CRATER.

5.1 CRATER architecture

This section describes a prototypical implementation of CRATER framework. Figure 5.1 shows the execution view of CRATER's architecture. In the following subsections I will explain in more details the components included in that figure.

5.1.1 Knowledge base modelling

Knowledge base can be modelled in many forms. Due to the selected case-base reasoning implementation [2], the knowledge base is modelled as *Tab Separated Values* (TSV) file. This file hold the cases that are ready to be adaptation response. Figure 5.2 illustrates how the knowledge base looks like. The cases in the knowledge base can be defined by the domain expert in addition to the newly learned cases. In the experiment I will start from empty knowledge base.

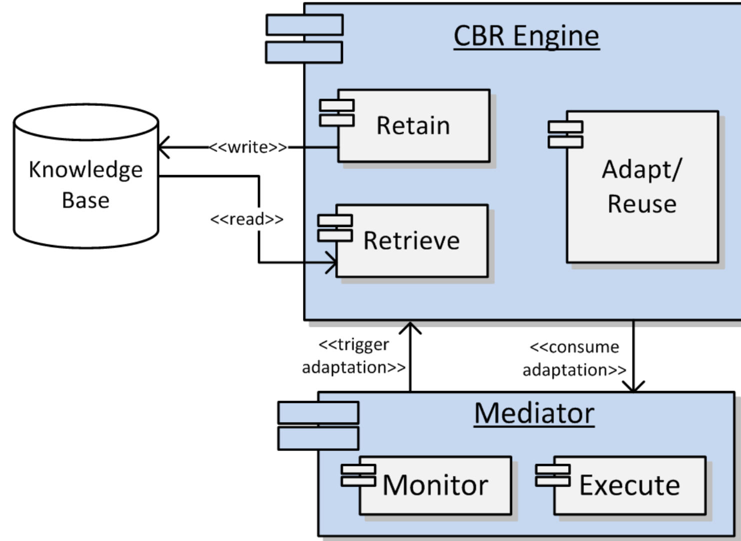


Figure 5.1: CRATER's Architecture: Execution View

5.1.2 CBR engine

CRATER utilizes the Case-based reasoning artificial intelligence technique. CRATERS uses CBR as an adaptation engine as described in Chapter 4. To that end, I used an open source CBR implementation [2] that provides basic functionalities of case-based reasoning like case retrial and similarity calculations. The component Retrieve in Figure 5.1 is implemented by [2]. The two other components, *Adapt/Reuse* and *Retain* is an addition by me to the work [2] and explained in the following subsections.

5.1.2.1 Adapt/Reuse component

This component is responsible for providing the adaptation response for the execute and retain components. The process of this component works as follow:

- The retrieval component is responsible for returning the similar cases as described in Chapter 4 that satisfies the β condition and initialize the Qualified Adaptation Frame.
- If the qualified adaptation frame is not empty, then this component returns the case with the highest expediency.
- Otherwise, due to empty knowledge base or not satisfactory case exist in it, the adaptation is generated by a First fit heuristic algorithm 2

2	Int	Int	Int	Int	Int	Int	Int	Int
3	3	2	3	3	3	2	2	1
4	3	3	2	3	3	1	1	3
5	3	2	3	2	2	2	2	3
6	4	2	2	2	4	2	1	4
7	4	2	1	2	4	2	1	4
8	3	3	2	2	2	1	2	1
9	3	2	3	3	4	2	1	4
10	3	2	2	1	3	1	2	3
11	2	3	3	2	4	2	2	4
12	4	2	1	2	1	2	2	3
13	3	2	1	3	1	2	2	3
14	2	2	2	3	3	2	1	3
15	3	2	1	2	2	1	1	4
16	4	3	2	1	2	1	2	2
17	3	2	1	2	1	2	1	4
18	4	2	1	3	1	1	2	3
19	3	2	1	3	1	1	2	2
20	4	3	3	2	2	2	2	3
21	2	1	1	3	2	1	1	4
22	2	3	2	2	3	2	1	4
23	3	3	2	2	1	2	2	2
24	4	1	1	3	2	1	2	2
25	4	1	2	3	3	1	1	2
26	4	2	3	2	2	1	1	2
27	4	2	2	3	3	1	1	2
28	2	2	1	2	2	2	1	1
29	2	3	2	3	2	1	1	4
30	3	2	2	2	3	1	1	3
31	3	3	2	2	1	2	2	1
32	4	3	2	1	1	1	2	4

Figure 5.2: Knowledge Base Snippet

Figure 4.3 depicts this process as a flow chart.

5.1.2.2 Retain component

Retain is simply preserving a the new generated case in the knowledge base. Retain component has access to the knowledge base to save the new cases. It is obvious that no duplication occurs in the knowledge base because the retain process is limited only for cases which are generated constructively after the β condition is not satisfied in the QAF.

5.2 Adaptation request

As described in Chapter 4 adaptation requests are managed system's states that requites adaptation. In order to perform this in the experiment for the robot system, adaptation request are generated randomly. The generation process produces robot's states that requires adaptation i.e. robot states with overall utility less than or equals *0.5*. The generation process iterates the robots attributes and randomly select a value from its value spaces. After selecting a value for all attributes, the process estimates the utility of the generated case and return it if its utility is zero. Otherwise the process continues till finding a utility breaker case. The adaptation request generator component does also the functionality of monitor and decider as it can decide whether the robot's state violates its goals or not.

5.3 Monitoring

Monitoring functionality is implemented in the prototype by the adaptation request generator. This is because the adaptation request generator provides only robot states with utility less than or equal UT e.g. less than zero. This functionality is the same thing as the monitoring process as monitoring will issue adaptation process when it the robot state's utility is less than or equal *0.5*.

5.4 Main classes

Figure 5.3 illustrates the main classes and their interrelations used in the implementation of CRATER's. I will describe abstractly these classes in the following paragraph:

- *RobotDataSheet:* This class maintains the robot's value for each attribute. It has also some methods for retrieving the values of attributes for both generating the adaptation requests and adaptation generation.
- *RobotState:* This class is a centric one. It keeps the robot's attributes as a set of instances of class *RobotAttribute*.
- *RobotAttribute:* This class keeps the name, the value, the weigh and the type of attribute[1].
- *RobotUtiltiy:* This class calculates the utility of the robot i.e. using *RobotState* object. Each utility is represented by the class *UtilityFunction*.
- *UtilityFunction:* This class represents one utility of the robot. Each instance of this class has name, value and weight.
- *Configuration:* This class saves the values for CRATER e.g. β, adaptation requests number, utility threshold and some other values for the CBR engine.
- *AdaptationController:* This class is responsible for doing the adaptation process. It uses the *QualifiedAdaptationFrame* class for processing the retrieved cases from the knowledge base and *FirstFitHeuristic* class which is responsible for doing the constructive adaptation.
- *ReturnCaseObject:* This class is an extension of the class *RobotState*. It is just a *RobotState* with extra attribute for similarity value. This similarity is returned form the retrieval process and is essential for the case expediency calculation.
- *Retain:* This class is used by *FirstFitHeuristic* class and is responsible for saving the new generated cases in the knowledge base.

5.5 Development tools

I used Java Development Kit (JDK) [3] version 1.7 for developing the application. The CBR implementation [2] is also a Java application. I also used NetBeans IDE [4] as development environment.

[1] Attribute types are described in Table 4.5

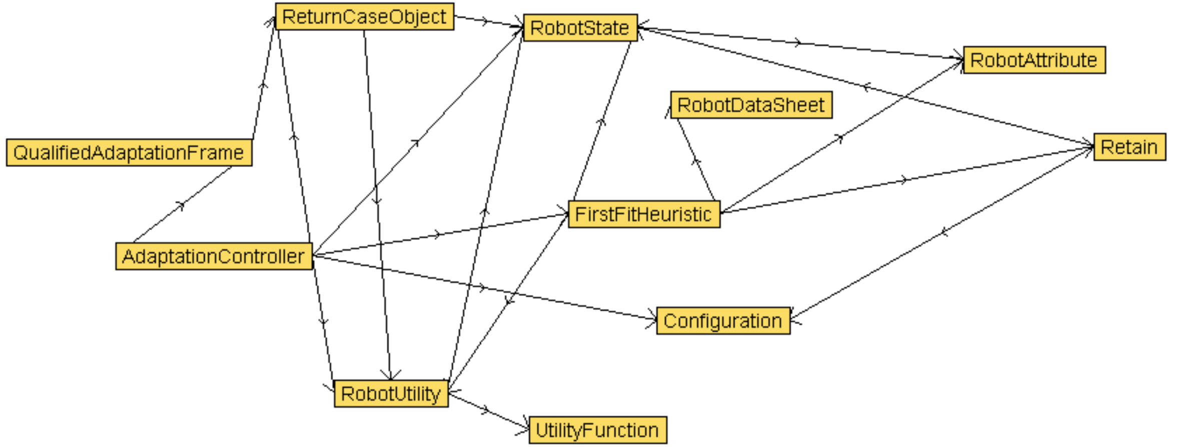


Figure 5.3: Main classes of CRATER

5.6 Uncertainty

As discussed in Chapter 4 uncertainty is represented in robot's attribute values as '?'. This uncertainty could be due to the unknown values or the absence of values at all. Figure 5.4 illustrates some basic classes used to realize uncertainty. These classes contains:

- *UncertainStateGenerator:* This class is responsible for randomly generating robot states that contain some uncertain values with the help of class *UncertainRobotDataSheet*. This class represents *uncertainty-injector*.
- *UncertainRobotState:* This class depicts a robot state that has a uncertain values. This class is centric one because it manipulates a list of states that represent possible states derived from the uncertain state. In addition it has the calculations for μ, Θ and η.
- *UncertainCaseAnalyser:* This class decompose the uncertain state into all possible certain states for next steps processing and it is used by the class *UncertainRobotState*.
- *UncertainCraterController:* This class is an extension of class *AdaptationController* described above. It provides the mechanism of adaptation in the presence of uncertainty.

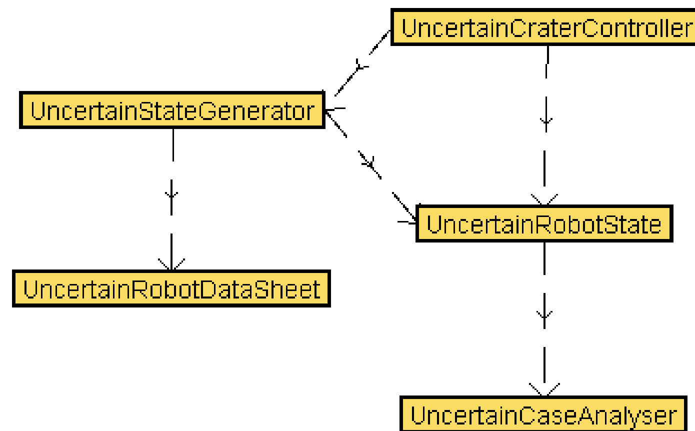


Figure 5.4: Uncertainty handling classes

5.7 Summary

This chapter showed the prototypical implementation overview of CRATER. This chapter described also how every component of CRATER is implemented. The implementation presented in this chapter included how the adaptation requests were generated randomly. Development tools and class diagrams for the implementation were presented in the context of implementation explained in this chapter.

Chapter 6

Experiment and Results

This chapter is intended to provide the experimental evaluation as proposed in the research method in Section 1.2. This chapter provides information about the validation and evaluation of CRATER in the light of the motivating example described in Section 4.1. This chapter starts with Section 6.1 that describes the experiment settings and continues with Section 6.2 that gives information about the derivation process of the metrics used for the evaluation and validation process. Section 6.3 embraces the results obtained form the experiment with extensive figures and information. This chapter ends with Section 6.4 that discuss the experiment's internal and external validly.

6.1 Experiment setup

The validation of CRATER is done in a binary validation paradigm. In order to validate CRATER, an experiment is conducted based on the motivating example described in Section 4.1 and implemented as explained in Chapter 5. The experiment was performed under Windows 8 (x64) machine with 4 GB of RAM and CPU Intel CORE 2 Duo (P7750) 2.26 GHz.

6.1.1 Design decisions

CRATER requires some configuration that needed to be set before working. These design decisions include:

- Utility threshold is 0.5. Choosing this value give the chance to show CRATER's ability in providing adaptation with greater utility.

- β is 90% unless otherwise stated. This value is suitable to show how CRATER construct cases and retrieve them from the knowledge base in the experiment.
- First fit heuristic is used in the implementation of the prototype. This is the provided implementation by the this prototypical implementation.
- In the experiment, CRATER starts with empty knowledge base which enhances the validation of the generative adaptation process.
- η is 85% unless otherwise stated.

6.1.2 Experiment nature

In order to perform the experiment, CRATER is subjected to seven successive runs. Each run contains 50 randomly generated adaptation requests as explained in Section 5.2. Seven runs were selected because they provide all important results and after that the results have insignificant effects.

6.2 GQM-based metrics

This section provides the related quality metrics for CRATER in the context of the motivating example described in Section 4.1. Adaptation-related metrics are formulated for evaluating CRATER based on GQM approach [39]. Goal-Question-Metric [39] is utilized in order to derive the related metrics for the validation purpose. In the following subsections software quality metrics are elaborated for evaluating CRATER.

6.2.1 Adaptation engine performance

Figure 6.1 shows the GQM sheet for identifying the performance CRATER adaptation engine. Two metrics are evolved and explained in the following subsections.

6.2.1.1 Adaptation remembrance

Remembrance implies making use of the previous performed adaptations instead of constructing the adaptation response from scratch each time. This property has a positive impacts on the performance of the adaptation engine. Remembrance is

G1	Purpose: Issue: Object: Viewpoint:	- Improve - The performance of - CRATER adaptation engine - 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	
Q3	What is the response time of the CRATER adaptation engine ?	
M2	Adaptation response time	

Figure 6.1: GQM Sheet for Adaptation Performance Goal

done using the knowledge base which contains the successful performed adaptations. Adaptation remembrance can be estimated by the following equation:

$$Adaptation\ remembrance = \frac{\#Adaptations\ retrived\ from\ KB}{\#All\ Adaptations} \tag{6.1}$$

6.2.1.2 Adaptation response time

The response time is traditional metric for performance evaluation. In CRATER the response time of the adaptation process is the elapsed time between receiving adaptation request and providing the adaptation response. The average response time for CRATER is estimated by the following equation:

$$Average\ Response\ Time = \frac{\sum_{i=1}^{n} Response\ Time(n)}{\#All\ Adaptations} \tag{6.2}$$

6.2.2 Adaptation expediency

Figure 6.2 shows the GQM sheet for providing an expedient and efficient adaptation response. I define expedient adaptation as the adaptation process that successes to rescue the managed system from undesirable states to desirable states in order to keep the keep the system goals satisfied including functional and non functional

G2	Purpose: Issue: Object: Viewpoint:	- Provide - An expedient and efficient - Adaptation Response - From the managed system view point
Q1	What is the adaptation expediency performed by CRATER adaptation engine ?	
M2	Adaptation Expediency	

Figure 6.2: GQM Sheet for Adaptation Expediency Goal

requirements. This is done by providing adaptation responses with utility greater than the UT. Adaptation expediency is quantified by the following equation:

$$Adaptation\ Expediency = \frac{\#Expedient\ Adaptations}{\#All\ Adaptations} \tag{6.3}$$

CRATER aims to provide an adaptation response with the highest possible utility. If CRATER always provides an expedient adaptation response each time then the adaptation expediency in Equation 6.3 equals one. However in some cases the managed system resources decrease overtime which affects the quality of the service provided by other components. CRATER has nothing to do in this case as CRATER has no authority on the resources of the managed system. Instead CRATER will provide an adaptation response with the highest possible utility. For example the power unit in the robot managed system described in Section 4.1 affects the video transmission unit. Thus the video transmission quality is governed by the available power such that if the available power is low then the video quality could not be very high. This affects the total utility of the robot and CRATER may not provide an adaptation with utility greater than UT because CRATER can not increase the available power in the robot and will only provide an adaptation response with the highest possible utility.

6.3 Results

This sections provides extensive results from the experiment explained in Chapter 4 and implemented in Chapter 5. The following subsections contains information regarding the response time and the expediency of adaptation.

6.3.1 Examples of adaptation

Table 6.1 illustrates two randomly selected adaptation done by CRATER one of them contains uncertain values. 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. The adaptation response for this disordered 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. The table also shows that the utility of the adaptation request is zero which is a utility breaker and CRATER managed to provide an adaptation response with utility 0.892 which is greater than zero and represents an accepted expedient adaptation response. The other adaptation requests holds uncertain value in the communication attribute. CRATER issued adaptation process for this robot state because the uncertain attribute, the communication, is uncertain and on possible values of it leads to zero utility. The value of communication attribute that causes zero utility is off which means that the robot is unable to establish connection with the remote centre. As a result CRATER issues an adaptation process that produces the adaptation response that assures that the communication is set with appropriate value enabling communication with the remote centre. 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 set off the communication. However CRATER did not chose this scenario because its utility is less than the utility of the chosen adaptation response.

6.3.2 Response time results

In this section I will provide the obtained results related to response time of CRATER. Figure 6.3 shows the average response time for seven successive run of CRATER starting from empty knowledge base and each run has 50 adaptation requests. It is clear that the average response time for any experiment is greater than the subsequent experiment. This is normal because most of adaptation requests in the first experiment were generated not retrieved form the knowledge base which is empty. The later experiments' average response time starts to decrease because the adaptation responses begun to be retrieved form the knowledge base which consumes less time than constructing adaptation responses. The average response time for experiment seven is the smallest among all experiments as the knowledge base became more

Table 6.1: Adaptation Samples

Attribute	Ad.Req.1	Ad.Res.1	Ad.Req.2	Ad.Res.2
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	Very 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**

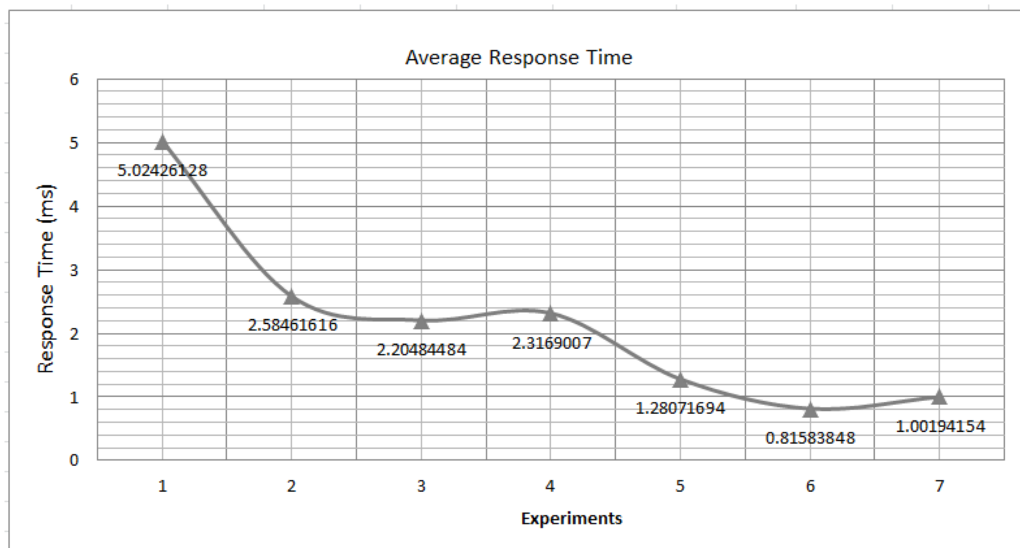
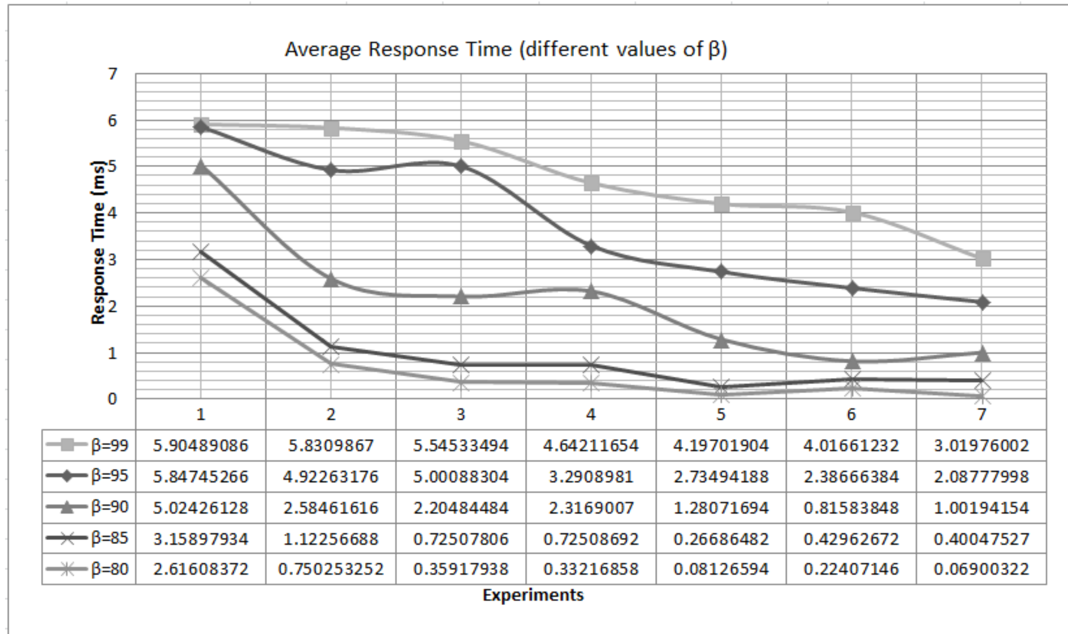


Figure 6.3: Average Response Time

sufficient and mature for providing adaptation response. However small numbers of adaptation responses could be constructed if the knowledge base fails to provide the required adaptation response.

6.3.2.1 β value effect on response time

Figure 6.4 shows the impact of the value of β on the average response time. If the value of β is high e.g. 99% or 95% this means that small number of cases in the knowledge base are selected in the qualified adaptation frame which leads to more constructively generated adaptation responses unlike small values of β e.g. 85% or 80% which consider more cases from the knowledge base. This leads to more retrieved adaptation responses from the knowledge base.



	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

Figure 6.4: Average Response Time: Different β values

6.3.2.2 Response time under uncertainty

Figure 6.5 shows the average response time for adaptation request with uncertain values with η equals 85% as stated in the experiment setup along with extra values

of η. All of the 50 adaptation requests depicted in Figure 6.5 has uncertain values to explain the effects of uncertainty in the performance of CRATER. All experiments in this figure has the same value of β which is 90%.

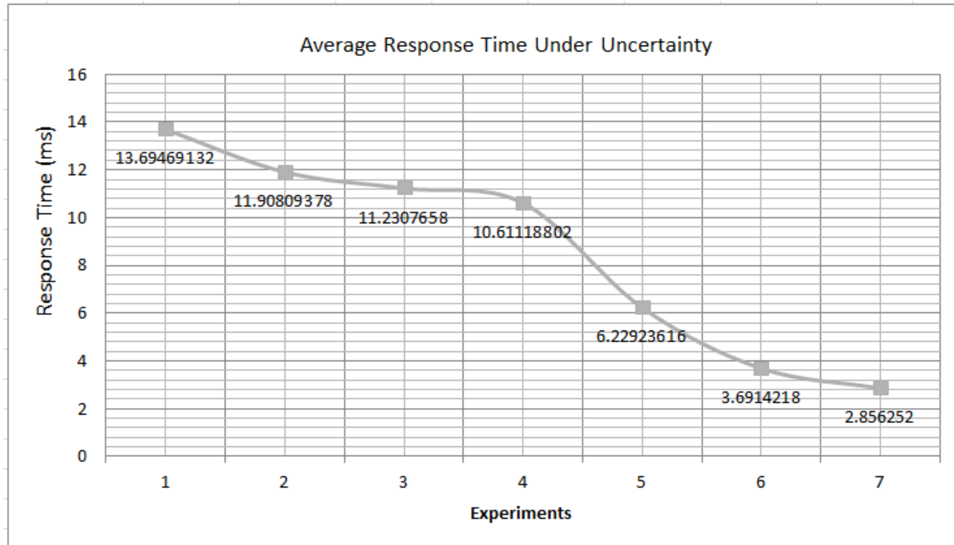


Figure 6.5: Average Response Time Under Uncertainty

Figure 6.5 shows that there is a slight increase in the average response time and it is more than what appears in Figure 6.3. This is normal because the adaptation requests with uncertain values requires more processing and analysing as discussed in Chapter 4 to find estimate μ and Θ.

6.3.2.3 η value effect on response time

Figure 6.6 shows the effect of η on the performance. If CRATER is configured with high value of η this leads to consider more uncertain adaptation requests. The less η is the less the average response time is.

6.3.3 Adaptation remembrance

As explained in Chapter 4 the adaptation response can be retrieved from the knowledge base or constructed if the knowledge base is not mature enough to provide the required adaptation. The remembrance measure finds the relation between the retrieved adaptation from the knowledge base and the total number of adaptations.

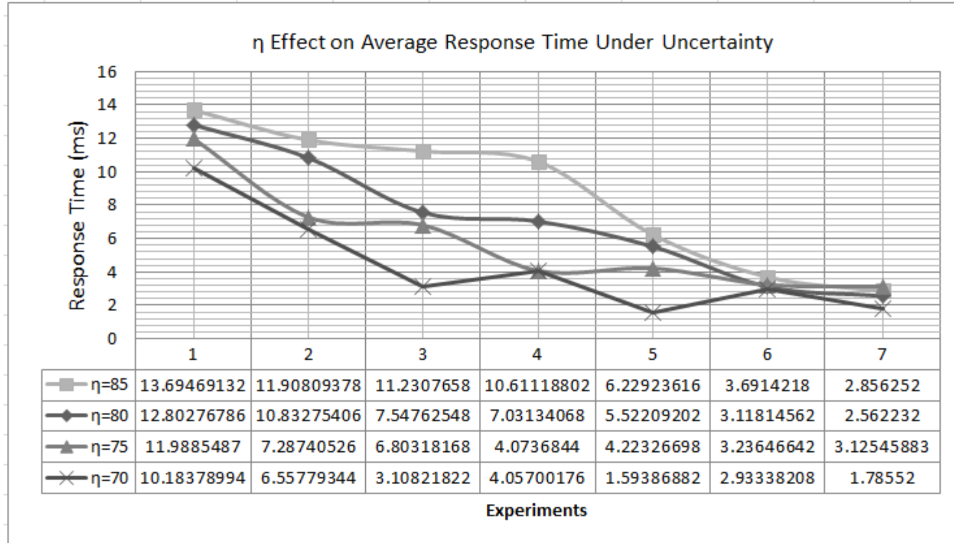


Figure 6.6: Average Response Time Under Uncertainty: Different η values

Figure 6.7 shows the remembrance of adaptation responses for seven successive experiments starting form empty knowledge base. In experiment number one the number of constructed adaptation responses is more than the retrieved which is logical as the knowledge base is empty. In later experiments the number of retrieved adaptation began to increase while the number of constructed adaptation responses began to decrease. This provides a positive impacts on the performance of the CRATER.

β affects the number of construed adaptation responses. To that end Figure 6.8 shows how β affects the number of constructed adaptation responses for seven successive experiments each of them has 50 adaptation response starting from empty knowledge base. It is clear from that figure that the more β value is the more constructed adaptation responses are. This means that choosing the value of β affects the performance of CRATER however it is still a design time decision.

6.3.4 Adaptation expediency

Adaptation accuracy is estimated by the adaptation expediency. Figure 6.9 shows the minimum, the average and the maximum adaptation expediency for seven successive experiments each of them has 50 adaptation responses starting from empty knowledge base. It is clear that CRATER succeeded in providing an expedient adaptation each

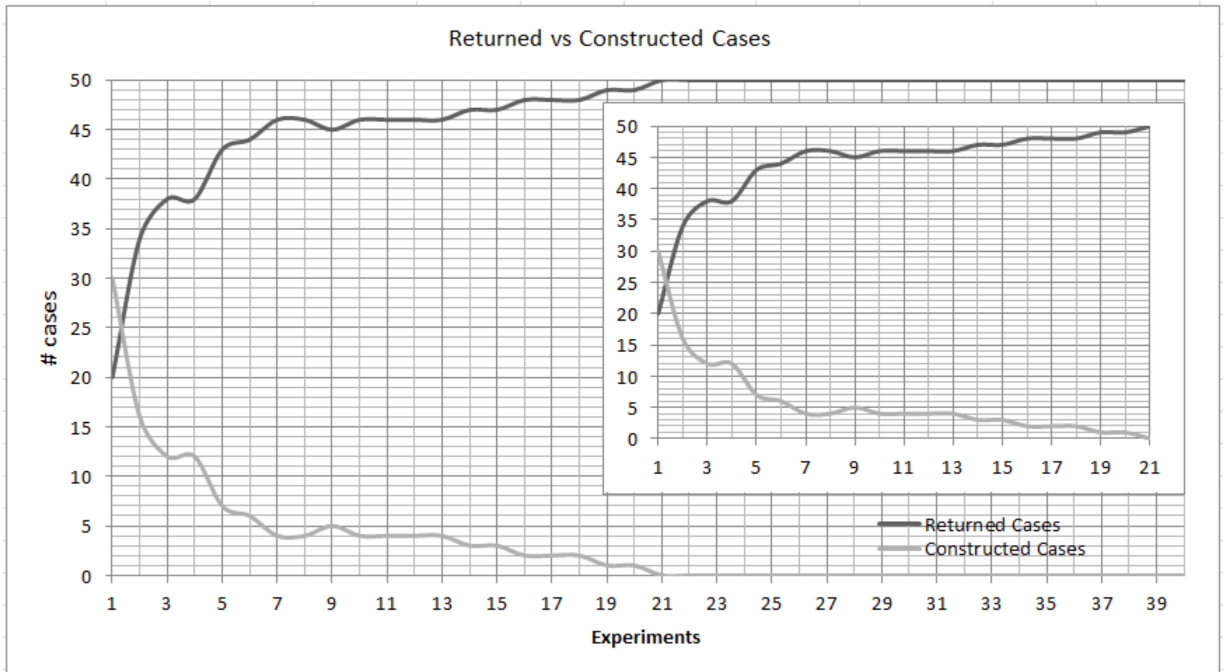
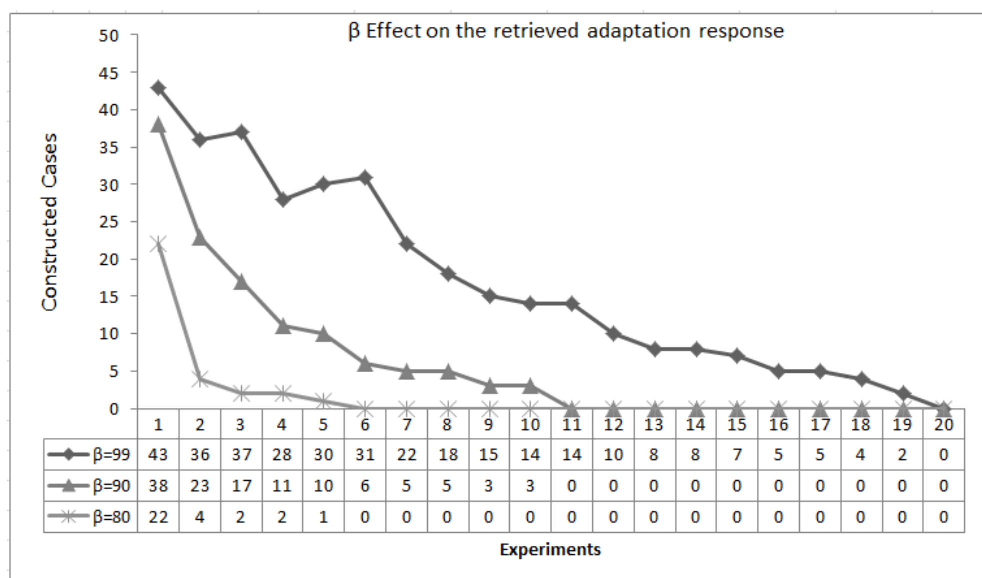


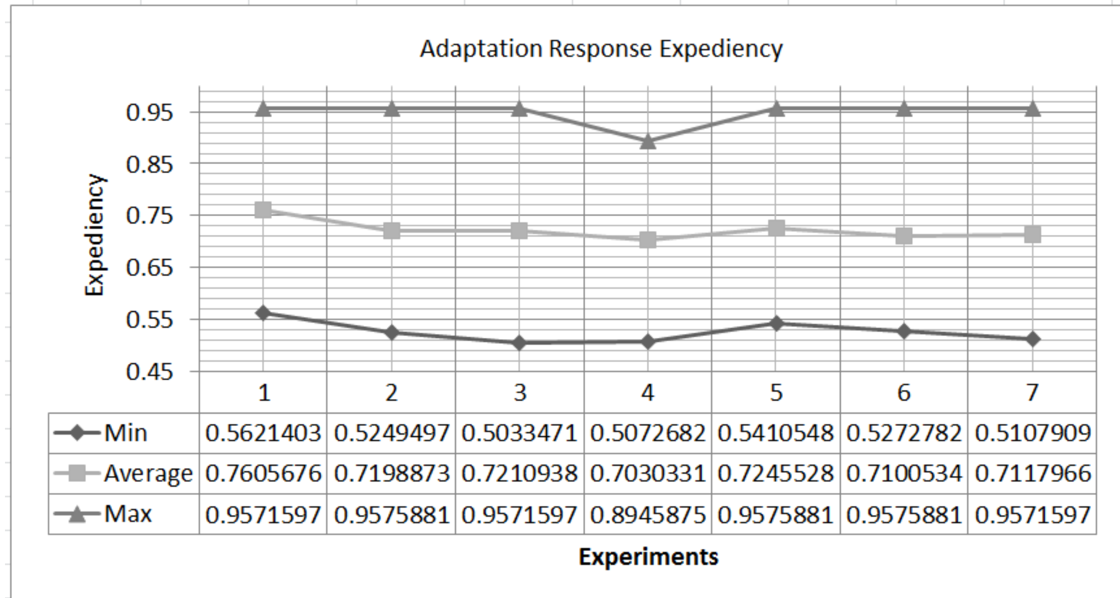
Figure 6.7: Adaptation Response Remembrance



	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

Figure 6.8: β Effect on the Adaptation Process

time with different expediency value depending on the nature of adaptation request itself.



	1	2	3	4	5	6	7
Min	0.5621403	0.5249497	0.5033471	0.5072682	0.5410548	0.5272782	0.5107909
Average	0.7605676	0.7198873	0.7210938	0.7030331	0.7245528	0.7100534	0.7117966
Max	0.9571597	0.9575881	0.9571597	0.8945875	0.9575881	0.9575881	0.9571597

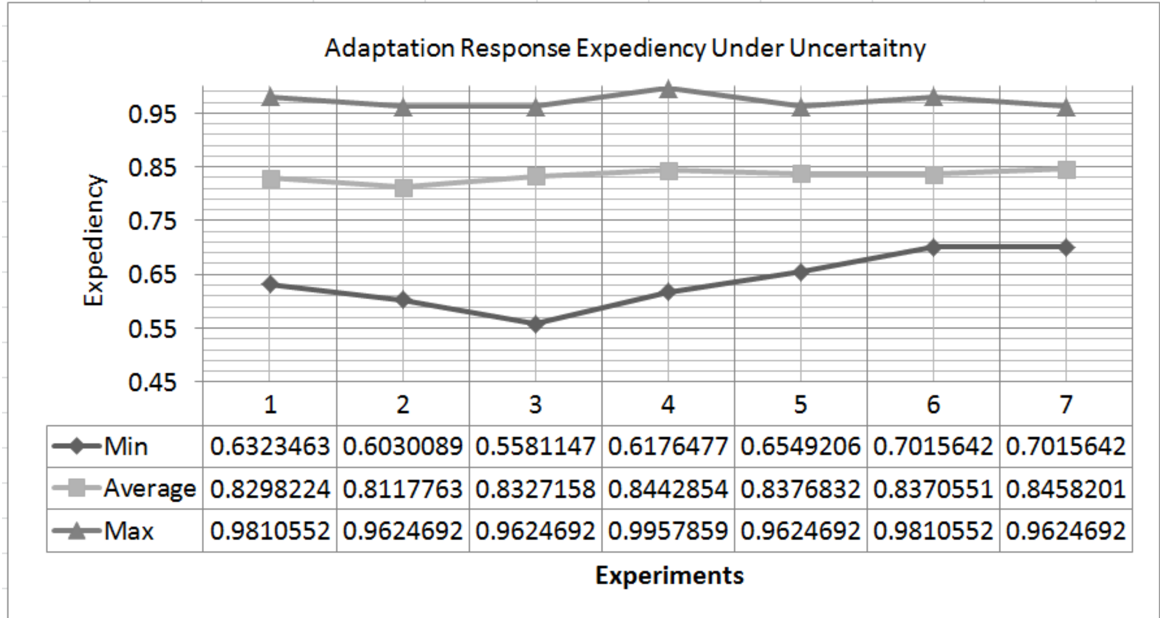
Figure 6.9: Adaptation Expediency

Figure 6.10 also shows the expediency of the adaptation process for adaptation requests all of them contains uncertain values. The figure explains that CRATER is in a position to work under uncertain situation and provide an effecient adaptation in terms of adaptation expediency.

6.3.5 Results discussion and research evidence

The primary goal of this thesis is to construct an adaptation engine for self-adaptive software systems which is accompanied with a research evidence. The research evidence for this thesis is to provide an *empirically evaluated evidence* regarding the goals of CRATER. In order to realize this evidence, a binary validation paradigm of the experiment is used to validate **CRATER's testable goals.**
As described in Section 1.1 and in the GQM sheets in Figure 6.1 and Figure 6.2, the goals of this experiment are:

1. G1: *Enhance the performance of the adaptation process by remembrance.* The success criteria used to binary-validate this goal is to test if CRATER managed



	1	2	3	4	5	6	7
Min	0.6323463	0.6030089	0.5581147	0.6176477	0.6549206	0.7015642	0.7015642
Average	0.8298224	0.8117763	0.8327158	0.8442854	0.8376832	0.8370551	0.8458201
Max	0.9810552	0.9624692	0.9624692	0.9957859	0.9624692	0.9810552	0.9624692

Figure 6.10: Adaptation Expediency Under Uncertainty

to retain a successful adaptation for later reuse and how this reuse affects the performance of CRATER.

2. G2: *Provide an effective mechanism for handling uncertainty.* The success criteria used to binary-validate this goal is to test if CRATER managed to provide an expedient adaptation response under predefined level of uncertainty η.

In the following paragraphs, empirical evidences for validating the aforementioned testable goals will be presented. These evidences are build upon the results presented in Section 6.3.

The remembrance rate of the cases as shown in Figure 6.7 increases overtime which enables CRATER to reuse cases stored in the knowledge base. This was clear from that figure as the average returned cases from the knowledge base is *46.85* cases versus construction rate of *3.15* cases out of 50 case. 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 which is 6.3% were generated constructively. This of-course affects the performance of CRATER as constructing new adaptation responses consumes more time than retrieving it from the knowledge base. The response time of CRATER 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 of CRATER for the performed seven runs of the experiment is (2.175 ms) based on Figure 6.3.

The uncertainty handling in CRATER is tested by identifying the adaptation expediency. Based on results shown in Figure 6.10, 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. Under these information CRATER managed to provide an expedient adaptation for all adaptation requests as shown in Figure 6.10.

6.3.6 Results conclusion

After the previous detailed results it is obvious that CRATER provides the adaptation mechanism in accepted manner in terms of (1) Response Time (2) Adaptation space utilization (by remembrance) and (3) Uncertainty diminution with the following constructive aspects:

- Developers need not to explicitly provide predefined system's ideal operation states and configurations. Instead, modelling utility functions for system's goal is enough for CRATER to operate and provide the results shown in the previous sections. CRATER is able to operate even there is no cases in the knowledge base. However, the developer can provide a starter cases with the help of the domain expert.

- CRATER provides an effective mechanism to overcome the problem of big operating states. CRATER memorizes the previously adapted scenarios for later using which has a positive impacts on performance.

- CRATER can operate under uncertainty that hinders the adaptation process. This represents a strong point over traditional solutions.

In the next section I will provide some threats that could affect the results and my conclusion.

6.4 Experiment validity

In this section I provide the threats to experiment external and internal validity.

6.4.1 Internal validity

The threats to internal validity in the experiment include:

- In reality the adaptation requests will be sent directly from the monitoring component during the monitoring process of the managed system to the adaptation engine. In the experiment conducted in this thesis 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.
- Another threat to internal validity could be the implementation of CBR engine. Different CBR implementation could provide slightly different results particularity in terms of response time even though the chosen CBR implementation [2] shows acceptable performance.

6.4.2 External validity

The generalization of my results could be affected by the chosen domain such that if CRATER is utilized in a different domain other than robotics. This could be figured out if I utilize CRATER in a different domain however I expect no major differences in the results.

6.5 Summary

This chapter presented details about the experiment, results and evaluation. This chapter presented also the GQM quality metrics used to evaluate CRATER. The experiment setup and the results of the experiment were described in details in this chapter. A validation and empirical evidence were introduced to validate CRATER. This chapter concluded with the experiment validity.

Chapter 7

Conclusions

Within this master thesis I presented CRATER, a framework for constructing an adaptation engine for self-adaptive software systems, which has some advantages over the existing solutions. The main contribution in this thesis was to provide an efficient adaptation mechanism that considers the number of possible adaptations along with uncertainty handling.

Before providing the information regarding how CRATER works I provided some basic information for the reader to understand the context of the thesis. After that I provided the related work to this thesis then started explain the idea of CRATER and how it works. After that I conducted an experiment to evaluate and validate CRATER's performance investigating its outcomes. The main challenges that CRATER managed to overcome were:

1. The mechanism of remembrance of the previous successful adaptations by storing them in the knowledge base for later reuse which reduces the time required to provide the adaptation response. This saved CRATER from doing the same computation if it receives the same adaptation request more than one time.

2. Handling uncertainty that appear in the adaptation process that hinders the adaptation process and leads to unrealistic adaptations. The experiment showed that CRATER is able to perform adaptation process under uncertainty.

3. The previously two challenges has been solved with a noticeable accepted performance in term of adaptation engine response time.

This thesis was supported with an empirical evaluation evidence described in Chapter 6. This empirical evaluation and evidence of CRATER bestows a trustworthy

approach for realizing self-adaptive software systems.

7.1 CRATER merits and limitations

The idea behind CRATER is founded based on two connected realities. On one hand the self-adaptivity property is realized in software systems by emulating the closed autonomic control loop firstly proposed in [15]. This emulation guarantees the automaticity of the system. On the other hand Case-Based Reasoning (CBR) life cycle, which was explained in Section 2.3.2, fits well for this emulation. In this section I will conclude, in the light of results in Chapter 6, with the merits and limitations of CRATER. The merits of CRATER includes:

7.1.1 CRATER merits

- *Operating space:* CRATER provides an effective mechanism to overcome the problem of big operating states. Firstly and thanks to CBR the existence of knowledge base represents a basic advantage for saving the best operating states of the managed system. These saved states, as cases in the knowledge base, can be exploited and reused during system runtime. These cases saves the time used to generate the adaptation response each time by learning new cases. Moreover integrating the utility functions provides an advantageous solution for representing the system goals and due to them the generating process of adaptation request becomes more directed and faster to keep the knowledge base preserving only the cases with utility greater than UT. Hence ensuring that the retrieval process returns only efficient cases from the knowledge base.

- *Managed system changes:* As being an external adaptation engine, CRATER is helpful in case of having legacy system that needed to become self-adaptive. This is because the changes required to be performed in integrating CRATER with legacy system is relatively small and is limited to providing some interfaces for both monitoring and executing components.

- *Adaptation response source:* CRATER provides dynamic adaptation responses because of the adaptation process discussed in Section 4.4.3 and shown in Figure 4.2 provides a learning mechanism. This contributes not only in generating

new solutions if the existing ones are not sufficient but also saving them for later reuse.

- *Adaptation process initiation:* Even though CRATER adopts the reactive adaptation style, it can be extended easily to support proactive adaptation. The *Observer and Decide* component in Figure 4.2 can be elaborated to provide a proactive mechanism. CRATER's components function separately exactly like what adaptation process suggests. Each component has its own functionality which enables future extension.

- *Autonomy vs Human intervention:* CRATER manages to start from empty knowledge base and operates without human inference which is an advantage. However this do not hinders the domain expert to define and refine the knowledge base contents. The Human-In-The-Loop principle can easily be applied in CRATER as a result of having the persisted knowledge base that can be changed at any time.

- *Uncertainty handling:* CRATER handles uncertainty which is one of the most decisive challenges in the self-adaptive software system field. The similarity measures that are the basic of CBR plays an important role in diminution of uncertainty and reduce its effects on the adaptation process particularity for unknown and missing values.

7.1.2 CRATER limitations

CRATER could suffer if the target managed system's adaptation related attributes do not have a predicted possible values. CRATER's mechanism is built upon a defined set of attributes values. Also the prototypical implementation may be improved to provide better performance particularly if the CBR implementation [2] is substituted with another one 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 CRATER used 8 attributes for the robot system. A problem could be raised if the managed system has more attributes with more possible values. This could affect the performance of CRATER. In this case an offline indexing for the knowledge base can be performed to enhance the performance of the retrieval process.

7.2 Prospective and vision

Applying CRATER on different case studies will provide an good indication regarding the applicability of it. It would be good if CRATER is applied in different domain e.g. information system domain applications, in order to deeply investigate its applicability. The next step is to prepare two publications the first will be about CRATER and the second about how the utility functions are utilized in case-based reasoning and applied in the CBR life cycle. Then finding a corresponding software engineering conference and case-based reasoning conference to submit them.

Bibliography

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

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

[3] Java development kit. www.oracle.com/technetwork/java/javase/.

[4] Netbeans ide. http://netbeans.org/.

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

[6] D. W. Aha. Case-based learning algorithms, 1991.

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

[8] R. Asadollahi, M. Salehie, and L. Tahvildari. Starmx: A framework for developing self-managing java-based systems. pages 58 –67, 2009.

[9] G. Bertolotti, A. Cristiani, R. Lombardi, M. Ribaric and, N. Tomas andevic and, and M. Stanojevic and. Self-adaptive prototype for seat adaption. In *Self-Adaptive and Self-Organizing Systems Workshop (SASOW), 2010 Fourth IEEE International Conference on*, pages 136 –141, 2010.

[10] B. Bontchev, D. Vassileva, B. Chavkova, and V. Mitev. Architectural design of a software engine for adaptation control in the adopta e-learning platform. In *Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, CompSysTech '09, pages 24:1–24:6. ACM, 2009.

[11] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. 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. ACM, 2009.

[12] S.-W. Cheng and D. Garlan. Handling uncertainty in autonomic systems, 2007.

[13] S.-W. Cheng, D. Garlan, and B. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, SEAMS '06, pages 2–8. ACM, 2006.

[14] M. Derakhshanmanesh, M. Amoui, G. O'Grady, J. Ebert, and L. Tahvildari. Graf: graph-based runtime adaptation framework. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 128–137, 2011.

[15] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. 2006.

[16] A. Elkhodary, N. Esfahani, and S. Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 7–16, 2010.

[17] N. Esfahani, E. Kouroshfar, and S. Malek. Taming uncertainty in self-adaptive software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 234–244, 2011.

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

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

[20] M.-H. Karray, C. Ghedira, and Z. Maamar. Towards a self-healing approach to sustain web services reliability. In *AINA Workshops'11*, pages 267–272, 2011.

[21] N. Khakpour, R. Khosravi, M. Sirjani, and S. Jalili. Formal analysis of policy-based self-adaptive systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2536–2543. ACM, 2010.

[22] D. Kim and S. Park. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on*, pages 76 –85, 2009.

[23] H. Liu, M. Parashar, and S. Member. Accord: A programming framework for autonomic applications. *IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Engineering Autonomic Systems, Editors: R. Sterritt and T. Bapty, IEEE Press*, 36:341–352, 2006.

[24] D. McSherry. Diversity-conscious retrieval. In *Proceedings of the 6th European Conference on Advances in Case-Based Reasoning*, ECCBR '02, pages 219–233. Springer-Verlag, 2002.

[25] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malex, and J. a. P. Sousa. A framework for utility-based service oriented design in sassy. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, WOSP/SIPEW '10, pages 27–36. ACM, 2010.

[26] A. Metzger. Towards accurate failure prediction for the proactive adaptation of service-oriented systems. In *Proceedings of the 8th workshop on Assurances for self-adaptive systems*, ASAS '11, pages 18–23. ACM, 2011.

[27] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, pages 44 –51, 2009.

[28] N. C. Narendra and U. Bellur. A middleware for adaptive service orientation in pervasive computing environments. In *Proceedings of the 5th International Workshop on Middleware for Service Oriented Computing*, MW4SOC '10, pages 19–26, 2010.

[29] V. Pareto. Cours d'economie politique. *F. Rouge, Lausanne*, 1896.

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

[31] M. Richter and A. AAmodt. Case-based reasoning foundations:the knowledge engineering review. *The Knowledge Engineering Review, Vol. 20:3, 203207*, 2006.

[32] M. M. Richter and S. Wess. Similarity, uncertainty and case-based reasoning in patdex.

[33] M. Salehie and L. Tahvildari. A quality-driven approach to enable decision-making in self-adaptive software. In *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pages 103 –104, may 2007.

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

[35] M. A. S. Sallem and F. J. 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)*, ARM '06, 2006.

[36] J. W. Schaaf. Fish and shrink. a next step towards efficient case retrieval in large scaled case bases, 1996.

[37] A. Stahl. Learning similarity measures: A formal view based on a generalized cbr model. pages 507–521. Springer, 2005.

[38] A. Sthal. *Learning of Knowledge-Intensive Similarity Measures in Case-Based Reasoning*. PhD thesis, 2003.

[39] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach. Goal question metric (gqm) approach. In *Encyclopedia of Software Engineering*. John Wiley and Sons, Inc., 2002.

[40] T. Vogel and H. 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, New York, NY, USA, 2010.

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

[42] S. Wess, K. dieter Althoff, and G. Derwand. Using k-d trees to improve the retrieval step in case-based reasoning. pages 167–181. Springer-Verlag, 1993.

[43] W. Wilke and R. Bergmann. Techniques and knowledge used for adaptation during case-based problem solving, 1998.

[44] Y. Wu, Y. Wu, X. Peng, and W. Zhao. Implementing self-adaptive software architecture by reflective component model and dynamic aop: A case study. In *QSIC'10*, pages 288–293, 2010.

[45] Q. Yang, J. Lü, J. Li, X. Ma, W. Song, and Y. Zou. Toward a fuzzy control-based approach to design of self-adaptive software. In *Proceedings of the Second Asia-Pacific Symposium on Internetware*, Internetware '10, pages 15:1–15:4, 2010.

[46] H. Ziv, D. J. Richardson, and R. Klsch. The uncertainty principle in software engineering, 1996.