REUSABLE ADAPTATION COMPONENT FOR SELF-ADAPTIVE SYSTEMS

KISHAN KUMAR GANGULY
BSSE 0505

A Thesis
Submitted to the Bachelor of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

BACHELOR OF SCIENCE IN SOFTWARE ENGINEERING

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

REUSABLE ADAPTATION COMPONENT FOR SELF-ADAPTIVE SYSTEMS

KISHAN KUMAR GANGULY

Approved:

Signature *Date*

Supervisor: Dr. Kazi Muheymin-Us-Sakib

To *Sankar Kumar Ganguly*, my father
whose endless inspiration has always kept me motivated

Abstract

Self-adaptive software development poses challenges in case of reusable adaptive logic generation, as the boundary between adaptation logic and business logic is often not clear. According to the separation of concern principle, the software adaptation logic and the business logic should be kept apart for reusability [1]. Developers have different perspectives in case of defining interfaces between these two parts. According to the context, self-adaptive systems change their behavior at runtime without any service interruption. As all these context changes cannot be defined at a time, to address these in the adaptation logic, it becomes more challenging. Due to these problems, the developed adaptation module does not become reusable. So, developers have to go through the same hardship over and over again for subsequent projects.

In this report, a reusable self-adaptive system design has been proposed. In this design, the feature, feature dependencies, metrics, utility functions that are equations representing metric threshold values and additional training features, which are predictors for training are collected. The knowledge base is generated by collecting metric values for different feature selection in a simulated environment. Then, training is performed which results in feature-metric relationship equations. When a need for adaptation is detected by analyzing the metric threshold values, an optimization problem is constructed to improve the overall utility of the system. The optimization problem results in a future selection that can be effected through effectors. To ensure the reusability of the subcomponents of

the adaptation component, design patterns were used to maintain separation of concern.

The proposed methodology was validated using a popular model problem named Znn.com. It was deployed in five servers which were balanced by a load balancer. Then, the code was analyzed statically to get the values of three reusability metrics namely *Lines of Code (LOC)*, *Message Passing Coupling (MPC)* [2] and *Lack of Cohesion of Methods 4 (LCOM4)* [3]. For reusability, it is seen that the proposed method contains 4367 LOC compared to 24891 LOC of Rainbow. For LCOM4, 86.15% classes have ideal value that is either 0 or 1. The MPC value is also low which is 3.046 in average. The system was put under high load to see if adaptation occurs and can improve the response time down to a threshold. It has been seen that the adaptation mechanism performs better than the system without any adaptation mechanism by gradually decreasing the response time. Thus, the proposed method seems to to be reusable and effective in terms of adaptation.

Acknowledgments

I would like to appreciate my supervisor Dr. Kazi Muheymin-Us-Sakib for his supportive guidance, continuous inspiration and candid advice. His support and help have inspired me to relentlessly work for producing the best quality work I can throughout the project. I would also like to thank my classmates of BSSE 5th Batch for their continuous motivation throughout the project.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the vastly dynamic operating environment now-a-days, software systems need to provide service according to the context in place. Self-adaptive systems are those which responds appropriately to the changing environment to provide better and relevant service to the users. Along with the growth of the self-adaptive system design approaches, reusability has become a concern [4, 5]. A reusable adaptation mechanism can save a lot of coding and testing time by providing a ready-made solution. In this chapter, the motivation behind this work and challenges have been discussed. This chapter also carries the research questions and contribution of the proposed self-adaptive system design mechanism. A section on how the report has been organized is mentioned at the end of this chapter.

1.1 Motivation

The motivation behind the work is the growing need for reusable and effective adaptation component, as seen from the literature [6]. Consider a scenario where the business logic of a system has been developed. As the adaptation component is complex, it takes time to develop. Besides, due to existence of a lot of adaptation mechanisms in the literature, the task becomes more difficult. This is because the literature provides no specific guideline in choosing an adaptation approach.

However, It is visible from the literature that many parts of the adaptation logic is common from domain to domain [7]. So, it would be better if these parts can be reused in this case. However, integrating these parts also pose challenges because integration requires some knowledge of the underlying code. So, the preferable choice is to generate the whole adaptation component and augment it to the business logic. The application specific parts of the adaptation component will contain hooks [7] to specialize. Besides, the reusable subcomponents inside the adaptation components will contain abstraction and flexibility to customize these as needed.

Another concern in generating a reusable adaptation component is to preserve the quality of adaptation while ensuring reusability. Along with the model being reusable, the adaptation logic also needs to be generic in nature. More specifically, the adaptation logic has to be expressed as a mathematical model because a mathematical model is mostly generic. An adaptation component that contains reusable logic and reusable subcomponents can help to reach the objective as mentioned in the previous paragraph.

Self-adaptive systems design methodologies based on architecture, control theory and machine learning have been proposed. MAPE-K (Monitor, Analyze, Plan and Execute with a Knowledge base) feedback loop [8], was proposed by IBM Corporation which was a blueprint for self-adaptive systems. Garlan et al. described the Rainbow framework [9] which was based on MAPE-K architecture. Rainbow used static strategies to perform adaptation at runtime. Although Rainbow aimed to achieve reusability by separating the adaptation component, the adaptation strategies or rules are system specific and need to be written separately for systems of different architectural styles. Cheng et al. proposed a technique where action was taken based on maximum utility [4]. However, it also had problems of aforementioned static condition-action rules. All these architecture-based approaches use an architectural model for analysis, and then perform changes to the

system with a translator. The requirement of a specific architectural model makes automated generation of reusable and easy-to-integrate adaptation component infeasible using state-of-the-art architecture-based approaches.

Control theory-based systems consist of a controller, plant (core system), sensors and references. Sensors receive the output of the system while controller compares it with the reference to compute an error value. This error triggers change in the parameter of the core system to bring its output as close as possible to goal. Filieri et al. proposed a six-step control theory-based approach where each step was described in details with mathematical formulations [10]. They mentioned that existing tools require system specific model so, these are not reusable across different systems. Y. Brun et al. discussed existing control theory-based tools and techniques in their paper [5]. They mentioned conversion of an existing system into a self-adaptive one through a tool as a research challenge.

Machine learning based approaches were also proposed to address the dynamic nature of self-adaptive systems. Kim D. et al. proposed a Q-learning based method where they used a Q-value measure for every reconfiguration of the system to choose the best one [11]. However, they did not consider reuse. Elkhodary et al. proposed the FUSION framework where relation between goals and variation points are learnt using a knowledge base [12]. The learnt relations are use to construct a optimization problem to choose a reconfiguration that satisfies all the goals. FUSION considered separation of concerns and provided opportunity for modular code generation. However, the scope of FUSION was to provide a feature oriented adaptation framework only. FUSION, as a tool, also depended on existing component control mechanism and model to code transformation for effecting the changes which may lead to the problems mentioned previously.

Some approaches based on component based software development have also been proposed. David et al. introduced a methodology [13] where the business logic needed to be implemented with the Fractal component model[14] and static

strategies were used to add or remove components to achieve adaptability. Although this technique considered reusability of adaptation logic, the requirement of following a specific component model caused the aforementioned problems. Wu et al. conducted a case study [15] on integrating dynamic Aspect Oriented Programming (AOP) and Fractal to provide reusability and adaptation capability to a system. It was seen that reusability was achieved. However, due to the problems of component models listed previously, this also does not provide a solution to the aforementioned problem.

It is evident from the literature that some of the approaches model the adaptation logic internally and so, fail to achieve reusability. Some approaches discuss external adaptation and focus on ensuring reusability through separation of adaptation logic from business logic. However, most of these either require following a specific model or achieves reusability of the adaptation logic only. It is also seen from the literature that no model aims to achieve a more granular level of reuse. This is why a self-adaptive system mechanism is needed which addresses more granular level reuse and provides effective adaptation logic.

1.2 Research Questions

As seen from the previous section, most of the existing literature on self-adaptive system design focus on adaptation mechanism rather than reuse. Although some of the works consider separation of concern, these require either system specific adaptation rules or a component model for integration. However, developers may not proceed towards writing specific adaptation rules for every cases or restructuring whole project following a component model due to limited time and amount of effort involved. Only a readily deployable adaptation component that can be reused and integrated without restructuring the codes to a specific model, can help to mitigate this problem. This leads to the following research question.

1. How can a reusable adaptation component be generated that adapts effectively in all systems?

   More specifically, this research question will be answered by the following sub questions.

   (a) How to model system specific adaptation rules for reuse?

   System specific adaptation rules can be modeled using features, metrics and metric thresholds. Features are variation points of a system and metrics measure some attributes of it. Goal violations can be captured using thresholds of the metrics that should not be exceeded for goal conformance. Metric values can be captured from the API that the developers of self-adaptive systems need to provide. This threshold and metric calculation component can be provided as input to the system so that different thresholds and metrics for different systems can be captured. Information about the features such as their names, package names and corresponding components are also given as input. After detection of goal violation, an optimization problem can be constructed considering metric and features to resolve conflict among goals. This problem can be solved to get a feature combination where maximum goal conformance is achieved.

   (b) How to devise a component control mechanism that can be reused across different projects?

   Interaction between business logic and adaptation component can be modeled with dynamic AOP or weaving. Using weaving, to perform swapping, the return value of an initiator method or factory method of a business logic class can be intercepted and another class can be returned. To remove a component, mock objects of the corresponding classes can be constructed and returned from the factory methods in the same way. The AOP based mechanism needs the presence of fac-

tory methods and requires coding to interface for automated component control. However, there may be systems where these have not been followed and thus refactoring is needed. To address this problem, a way to customize component control mechanism can be provided where developers can specify implementations of adding, removing or swapping components according to the design of the existing system. This customized component control will help to conform to the design of the existing system with minimal implementation rather than restructuring the whole code base.

1.3 Contribution and Achievement

The contribution of this work is the technique for designing reusable adaptation component for self-adaptive systems. From Section 1.2, it is evident that there are mainly two goals which are achieving systematic reuse by incorporating easy adaptation component generation and integration mechanism ,and providing an effective adaptation model. To address the first goal, the system specific information which are information about features, metrics, metric thresholds etc. are received as input from the user. The adaptation logic is a mathematical model that produces an optimization problem from a prediction model on goal violation. The effectors, automated or customized, help to execute the feature selection obtained from the solution to this problem. Thus, adaptation code is kept separated from the business logic codes. The components are modeled using design patterns to reach reuse at a more granular level.

The second question, that is to achieve an effective adaptation mechanism has been addressed by modifying the FUSION framework [12, 16] with additional training features and a knowledge base generation component. The additional training features can help to obtain a more accurate prediction model and the

knowledge base generation component helps to resolve the problem when no data is present for computing the prediction model. Thus, these two both increase the effectiveness of adaptation and create a more generic adaptation logic.

The proposed methodology was applied on Znn.com, a model problem provided by the Software Engineering for Self-Adaptive Systems community [17]. The system was deployed in five servers balanced by a load balancer. Three metrics namely *Lines of Code (LOC)*, *Message Passing Coupling (MPC)* [2] and *Lack of Cohesion of Methods 4 (LCOM4)* [3] were used to test the reusability of the approach. To test the effectiveness of adaptation, the system was put under high load to see if adaptation can bring the response time within a threshold. In case of reusability, the proposed method contains 4367 LOC compared to 24891 LOC of Rainbow. 86.15% classes have ideal LCOM4 value which is either 0 or 1. The MPC value is also low measuring 3.046 in average. The proposed adaptation method also performs better in the high load than the system without adaptation as it brings down the response time as soon as it rises. In this way, the proposed method has been validated to be reusable and effective.

1.4 Organization of the Report

In this section, the organization of the report has been shown to provide a roadmap to this document. The organization of the chapters in this report has been mentioned in the followings.

Chapter 2: The definitions and background information of self-adaptive system design have been discussed in this chapter.

Chapter 3: In this chapter, the existing works for self-adaptive system design have been presented is a structural way.

Chapter 4: This chapter contains the proposed methodology for the reusable

adaptation component design.

Chapter 5: The experimental setup, implementation and result analysis based on a case study have been provided in this chapter.

Chapter 6: This is the chapter that summarizes the whole report and highlights future work.

Chapter 2

Background Study

With growing complexity of software and uncertainty in the environment, Software Engineering has become complicated [5]. Due to many operating environments such as different operating systems, database systems, web servers etc., accurate prediction of software runtime behavior has become impossible. This is why it is not possible to state all the software requirements. specifically quality requirements [6] (for example, performance, security etc.), at the time of requirement specification. So, to make a software conform to changing requirements or goals at runtime, a mechanism is needed.

Self-adaptive systems are those which responds to context changes at runtime. For a few decades, there has been a shift towards developing systems in a self-adaptive manner to address the problem mentioned in the previous paragraph. The concepts and designs from self-adaptive systems have been applied to fields such as robotics, vehicle control, sensor system, signal processing etc. [18]. The growing need of self-adaptive systems has called for proper Software Engineering principles to develop these [6]. However, research on software engineering principles for these systems such as design, development, testing, reusability and modularity etc. is still in an initial phase [6, 19]. Recent literature [6, 19] shows that there are yet many challenges concerning self-adaptive system engineering,

specifically self-adaptive system design. This is why self-adaptive system design needs further attention.

2.1 Self-Adaptive System

Without interrupting their service, the systems which are able to change their behavior and functionality at runtime, according to the context in place, are known as *self-adaptive systems*. *Context* is the operational environment of a software. As mentioned in [20], information that can be used to describe the situation of an entity is called context, where entity can be any object (for example, user, application etc.) related to the situation. *Sensors* are used to collect information about the context. This information is used to analyze and take decisions that are executed by *effectors* which are interfaces that allow structural changes (component add, removal etc.) within a system [21].

The definitions of self-adaptive systems have been provided from numerous perspectives. Brun et al. defined that, *self-adaptivity* refers to the capability to adjust behavior at runtime [5]. The *"self"* [5] prefix indicates that the system decides and acts by itself dynamically. They also mentioned that human intervention in the form of policies for adaptation along with autonomic behaviors can improve adaptivity [5]. Salehie et al. stated that, a self-adaptive system changes its behavior only when a goal is violated or improved functionality is possible [22]. According to Esfahani et al., self-adaptive systems adapts at runtime to achieve *functional* or *quality-of-service* goals [12]. Krupitzer et al. mentioned that, a self-adaptive system modifies itself at runtime by adjusting *parameters or artifacts* [23].

From the definitions, it is noticeable that an adjustment of behavior is triggered as soon as a goal violation occurs. Although these goals can be divided into functional and non-functional ones, self-adaptive system community have focused

more on non-functional goals [24]. The definition by Krupitzer et al. also mentioned that effecting a change may include changing attributes which are known as configuration parameters. It may also include adjusting artifacts which may range from changing an architectural representation to performing component level changes. From the definition by Brun et al., self-adaptivity does not always refer to fully autonomic behavior. Manual intervention, specially human developed policies or rules are useful for generating a plan for adaptation. The definitions and discussions combined, lead to four functions for self-adaptive system which are monitoring, analyzing, planning and executing the plans.

2.2 Applications of Self-Adaptive Systems

As self-adaptive system design have been explored from various areas, it has been applied on different domains such as *sensor networks*, *intelligent infrastructure systems*, *manufacturing process*, *social service based-systems*, *transportation* and finally in *software industries* [24]. Generally, self-adaptive systems can be applied when a decision is necessary in the presence of an uncertain environment. The applications of self-adaptive systems from the mentioned perspectives have been given below.

2.2.1 Sensor Networks

Sensor networks have been used in military systems, health and manufacturing systems etc. [24]. As these systems are dynamic in nature, that is situation change occurs quite often, these expose an area for self-adaptive systems to explore. For example, *RoboCup-Rescue* [25] project was a disaster management robot simulation that used sensor network for surveillance purpose, as mentioned by Macas-Escriv et al. [24]. Here, the sensor network needed to have self-adaptivity. *Design, Monitoring and Operation of Adaptive Networked Embedded Systems (DEMANES)*

was a project that aimed at developing component-based frameworks and tools for developing self-adaptive systems. It was also mentioned that DEMANES would be applied to cooperating sensors at home [26].

2.2.2 Intelligent Infrastructure Systems

These are systems that support communication, clean water and other such physical utilities. Self-adaptive systems can be use to provide effective support in such varying environment. For example, *MULTIFORM* [27] is a project that can help to design such systems with feedback control [5].

2.2.3 Manufacturing Process

In a manufacturing industry, large processes are involved. It is necessary to exert enough monitoring and control over these processes to enhance quality [24]. As self-adaptive systems deal with monitoring and control [5], these can be used to develop effective manufacturing processes. For example, Scholze et al. described the application of self-adaptive systems in shoe manufacturing where the process in analyzed continuously to keep the process optimal by adjusting parameters such as pressure, temperature, speed etc. of the pump [28].

2.2.4 Social Service-Based systems

In social service systems, adaptation can help the decision making process. For example, health caring systems use intelligent network that provide the doctors the best data suited to the context to take a health-related decision [24]. Recently, Sarriot et al. performed case studies on municipal health systems in Bangladesh and found that these worked like complex-adaptive systems [29].

2.2.5 Transportation

Transportation system is dynamic by nature. So, self-adaptive systems can also help to analyze it [24]. National transportation systems consist of many divisions and so a complex network exists within these. This also happens for military systems. In this case, self-adaptive systems can help to take decisions within this variable complex network.

2.2.6 Software Industries

Antivirus of IBM use biological techniques for anomaly detection which resembles to self-adaptation monitoring process [24]. Besides, the field of robotics have always tried to use concepts from self-adaptive systems to build more responsive robots. For example, Denneberg et al. discussed *Open Software Concept for Autonomous Robots (OSCAR)* which consisted of four layers namely *command layer*, *execution layer*, *image layer* and *hardware layer* [30]. These layers corresponded to the functions of a self-adaptive system.

2.3 Self-Adaptive System Life Cycle Model

The life cycle of self-adaptive systems, discussed by Andersson et al. [31] covers the interaction between its development and operational state. The activities in these two steps are also known as *offline* and *online* activities respectively. The life cycle model has three stages which are *initial development*, *evolution and adaptation*, and *phaseout* [31]. On each stage, the offline and online activities interacts with one another which are known as *interaction points* [31].

Figure 2.1 shows interactions between development process and a running self-adaptive system. In the first figure, X axis represents time and Y axis represents development activity level. X and Y axis represents time and service level of a running self-adaptive system respectively in the second figure. At first in the

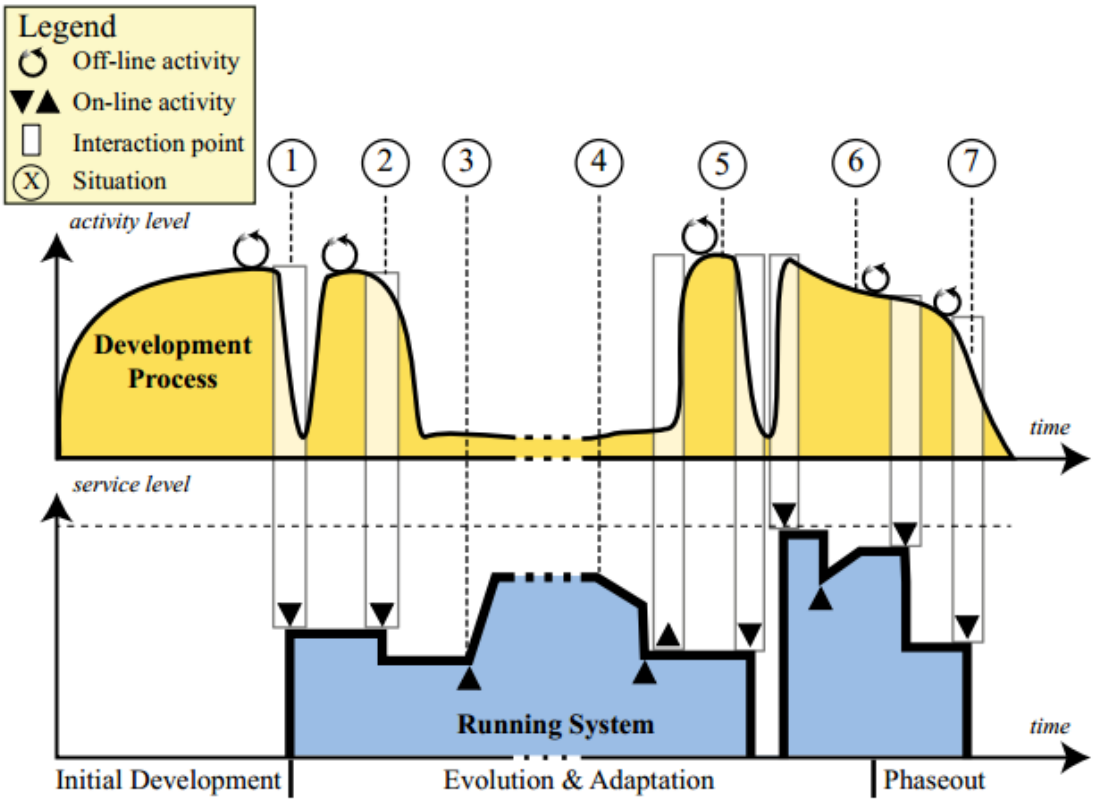


Figure 2.1: Timeline View of Development Process and a Running Self-Adaptive Software System (*Reprinted with permission from Springer, Copyright 2013, Springer-Verlag Berlin Heidelberg*)

initial development phase the system is developed offline and provided with some primary workarounds which can be alternatively used in place of one another. After initial development it is deployed in (1).

After deployment, as the system is running, developers continuously update business logic and add corresponding workarounds in the system through online updates. In (2), it is seen that one or more online updates may have caused a component fault which have negatively affected its service level.

In (3), a situation have occurred at runtime which have negatively affected the service level. The aforementioned workarounds are now used and the system adapts to a point where service level increases.

The system may not be able to use provided workarounds to increase its service level if an unknown fault occurs. As shown in (4), the system needs to be shut

down and developed following offline activities because online update may be too complex. The new context is considered and the fault is corrected offline. Then, another deployment occurs and the system continues online. This situation has been shown in (5).

At some point, workarounds that may be added to address the fault is identified and developed offline. Then, it is updated online as depicted in (6). Finally, the system leads to phaseout level when in future it is decided to be discontinued. (7) shows this phase where the system is shut down permanently and service level reaches zero.

2.4 Self-Adaptive System Design

To fully describe a system, its architecture, components, interaction between its components and its interface need to be specified. *Self-adaptive system design* states how adaptive systems are comprised from high level (architecture), how these high level components can be broken down into smaller components and interact to achieve adaptivity. Self-adaptive system design is a trending research topic in software engineering. As a result, numerous models [13, 9, 4, 32, 12] have been proposed on self-adaptive system design. *Architecture model* [9, 33], *component model* [13, 15], *machine-learning* [12, 11], *control theory* [32, 10], *software product line* [34, 35] and *design pattern* [36, 37] based models are the most prevalent ones, as seen from the literature. The following subsections introduce the concepts related to these design models.

MAPE-K architecture, which is a widely used blueprint for self-adaptive system is explained first. Then feedback-control and reinforcement learning is described which are related to control-theory and machine learning based approaches respectively. A brief discussion on software architecture and component models is presented, followed by a description of software product line and design patterns.

2.4.1 MAPE-K Architecture

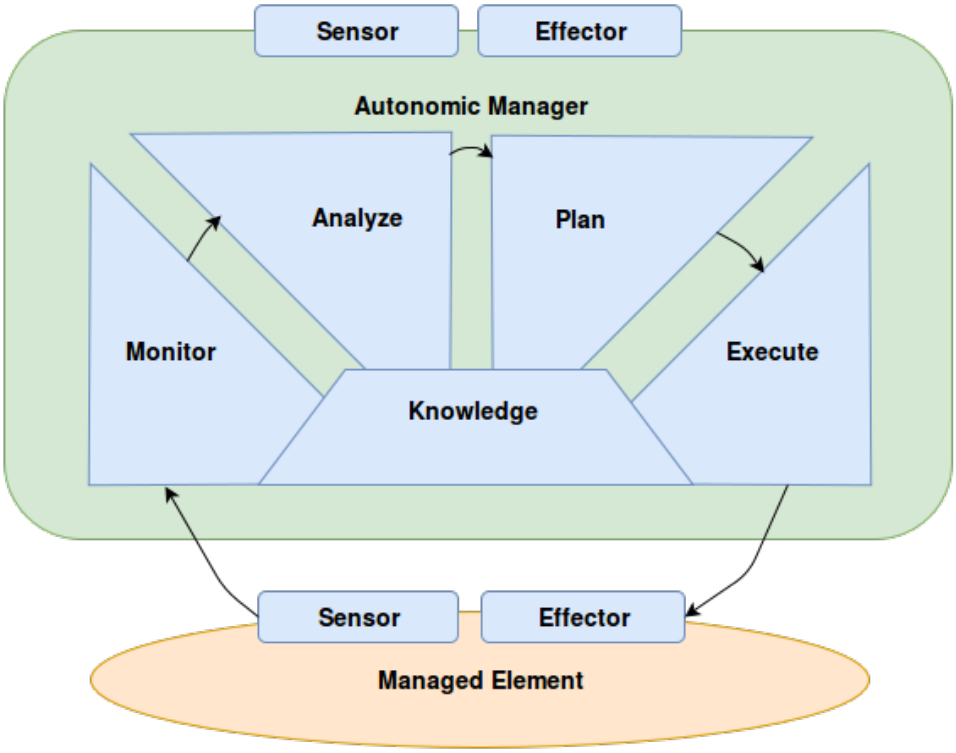


Figure 2.2: MAPE-K Feedback Loop by IBM

The *MAPE-K* architecture was proposed as a blueprint for an *autonomic manager*. According to IBM, an autonomic manager is a component that automates a management function and makes this function externally visible using an interface [21]. The architecture of this autonomic manager has been shown in Figure 2.2. The figure depicts that information captured by sensors are passed to *monitor* function and then it is inspected by *analyze* function. The analysis triggers the *plan* function and finally plans are executed through effectors from the *execute* function. Each part is described briefly in the following subsections.

1. Monitor: This function receives information from the sensor interface. The information is aggregated or organized in such a way that these correspond to analyzable symptoms. For example, collected information can be mapped to a metric (for example, performance) and passed to the analyze function.

2. Analyze: This determines if symptoms passed from the monitor indicate that a violation of the system goal has occurred. In this case, a logical change request for that particular symptom is passed to the plan function. For example, if performance degradation is detected, a change request indicating performance goal violation is issued.

3. Plan: The activities to bring back the system close to its goal are developed by the plan function. These activities can be as simple as a single command or complex such as a workflow consisting of multiple activities. In the example of performance degradation, it can construct a plan for disabling some memory consuming components that have been unused for a particular time period.

4. Execute: This function schedules and performs changes proposed by plan phase to the system. This changes are done through effector interface. Execute can also update the content of *knowledge base* [21].

5. Knowledge: Knowledge is the data used and shared between the four functions mentioned above. It includes policies, logs, historical information etc. A knowledge base can be pre-supplied or constructed by the autonomic manager. The historical information in the knowledge base needs to be updated regularly as it holds the patterns useful for predicting resources or symptoms.

The knowledge also can be divided into three types which are *solution topology knowledge*, *policy knowledge* and *problem determination knowledge* [21]. Solution topology knowledge consists of system configuration and component information which the plan function may use for choosing valid activities. Policy knowledge is used for deciding if changes can be deployed into the system. Problem determination knowledge consists of monitored and symptoms data that the loop may use to learn behaviors of adaptation.

2.4.2 Software Architecture

Architecture is the structure of a software system, where components are the building blocks. Architecture of a software is defined and designed in the architectural design phase. Various architecture-based adaptation techniques have been proposed in the literature. This is why discussion on software architecture, specifically *architectural models* and *architectural styles* is important. In the literature, software architecture has been defined from various perspectives. Shaw et al. defined architecture as an association of components and mentioned that subsystems comprise a system through *architectural operators* [38]. Pressman mentioned that software architecture is comprised of components and shows how different components interact to build an overall system [39]. He also defined *architectural design* to be the stage where software components, their properties and interactions are defined [39]. Architectural design helps to analyze and see a complete picture of a software before it is built and is an enabler for earlier design decisions [39]. In an architecture-based self-adaptive system, architecture design model is analyzed for goal violation and then updated according to adaptation decisions [9]. These updates are also executed on the system by the effectors.

Architectural design models can be constructed following specific patterns which are known as architectural styles [38]. These help introducing a consistent structure throughout a system which enables runtime changes such as component addition or removal. In the literature, architectural styles have been used to build reusable self-adaptive systems [9]. Architectural styles are implemented by *Architectural Description Languages (ADL)* [38]. ADLs are higher level languages that describe the architecture of a system in a structured way. ADLs describe the components, their interconnections, their interface, component roles and various other constituents of an architecture [38]. Most ADLs have both graphical and natural language syntax. For example, *xADL* [40] is a popular ADL that also supports modeling architecture visually through a tool called *ArchEdit* [40].

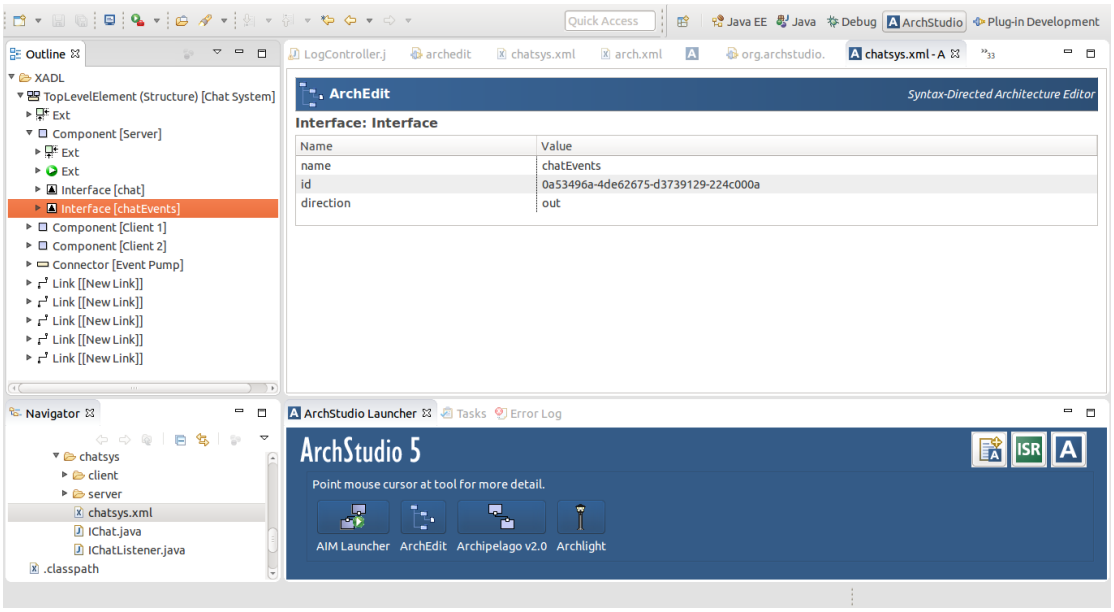


Figure 2.3: Interface of ArchEdit with xADL support

Figure 2.3 depicts the interface of ArchEdit with a model for a server to client chat project. Both graphical and natural language based ADLs have been used not only in architecture-based self-adaptive systems [9, 22] but also in machine learning [12] and component model [41] based ones for structural support.

In an architecture-based self-adaptive system, architectural models are used in the *goal management*, *change management* and *component control* phases, as mentioned by Kramer et al [42]. For example, in *FUSION* which is a famous model for self-adaptive system, Esfahani et al. used *Goal Modeling Environment (GME)* [43], *XTEAM* [44] and *Prism-MW* [45] tools for goal management, change management and component control respectively [12]. GME is a modeling tool for building complex architectural models. GME supports component hierarchy, component sets, connectors, references and component constraints. XTEAM is based on xADL [40] and provides support to combine different ADLs in a single model. Prism-MW is a middleware which supports automated transformation of architectural model to code. All these tools support *metamodeling* which is a technique for designing metamodel or schema of a model. A model conforms to a metamodel. FUSION [12] uses a metamodel in each of the three phases and any model of a sys-

tem that uses FUSION have to conform to this metamodel. It is mentionable that the language for this system specific model is known as *Domain-Specific Modeling Language (DSML)*. The DSML, its metamodel and model interpreter constitute the area called *Model Driven Engineering (MDE)*[46]. Recently, researchers have shown that MDE can be used to build effective self-adaptive systems [47].

2.4.3 Software Component Model

As software development is focusing more on maintainability and reuse, *Component Based Software Engineering (CBSE)* is getting more attention by Software Engineers. The unit of CBSE is a *software component* which is reusable, replaceable and independently deployable. Recent works on self-adaptive systems are focusing more on CBSE-based software development [13, 15]. This is why a discussion on CBSE is required. Software Component has been defined in the literature from various perspectives. Szyperski et al. defined a component as a unit of composition with third parties and interface as per contract, and a independently deployable entity [48]. Chaudron mentioned that a component is an unit of *"independent deployment, replacement, reuse and composition"* [49]. These definitions indicate that components are self-contained entities and these can interoperate with one another through appropriate interfaces. In Object Oriented Programming (OOP), a component is a set of classes [39].

A *component model* defines the standards for component composition [49]. This indicates that a component is a building block of a system that conforms to a specific component model. For example, *Fractal* is a component model that has been used in [13] and [15] for developing component based self-adaptive systems. Fractal supports two types of components which are *primitive* and *composite* components [50]. A primitive component consists of a single class. A composite component is a set of primitive components, hiding some behavior as a whole. Each component has a *controller* and *content* [50]. Controller exposes an inter-

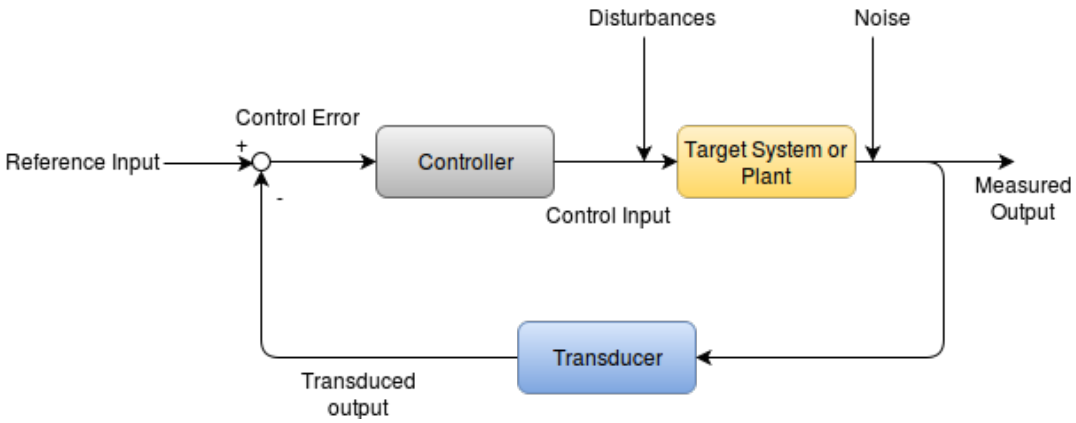


Figure 2.4: Block Diagram of Feedback Control Loop

face for the content. Content is the actual code of the component. Fractal supports *reflection* [50] where it can add, remove and inspect components at runtime. This can help to achieve *structural reconfiguration* (adding, removing or swapping components) [48] which is necessary for executors in self-adaptive system. Fractal component model has a large feature set, a complete discussion of which is out of scope of this report. Interested readers are refereed to the Fractal specification in [50].

2.4.4 Feedback Control

Feedback control is a process for regulating the output of a system towards an expected value. *Control theory* is a systematic approach for designing accurate and stable feedback control loop [51]. In a feedback loop, goals are provided as input and the output of the system are used as feedback. According to the feedback, parameters of the system are tuned to push it towards goal. From the definition of feedback loop, it is evident that feedback loop can be used to develop self-adaptive systems. From the literature, it is seen that approaches such as [32, 10] use control theory to design feedback loop for self-adaptive systems. Figure 2.4 depicts the structure of the feedback loop. The feedback loop consists of three components such as *target system or plant*, *controller* and *transducer* [51].

1. Target System or Plant: Target system is the one which is being controlled. Target system accepts some control inputs which represent values of some parameters (for example, size of a queue). By setting the parameter values, the system is tuned towards goal.

2. Controller: Controller exerts control to the plant through control inputs. Controller calculates the values of the control inputs based on both past and present values of control error.

3. Transducer: Transducer translates or converts measured output in a form which is comparable to reference input.

From Figure 2.4, five types of data are seen in the loop which are *reference input*, *control input*, *disturbance input*, *noise input* and *transduced output*.

1. Reference Input: This is the expected value of the output. For self-adaptive systems, this can be expected value of a metric, for example, response time of 0.5 ms.

2. Control Input: Parameters, the value of which can be adjusted at runtime are known as control inputs. For example, to improve performance, controller may take the decision to update a parameter relating to maximum number of servers.

3. Disturbance Input: Disturbances are changes that influence how control inputs interact with measured output.

4. Noise Input: A noise may change the measured output from the plant. Noise may occur at calculation time or data collection time.

5. Transduced Output: This is the transformed output passed through a transducer. For example, the transducer may normalize performance measures to make those comparable to reference performance values.

In case of a self-adaptive system, during adaptation, control outputs need to be predicted for choosing the best control inputs or parameter values. In a feedback control loop, the current output depends on previous outputs and control inputs [51]. For example, the current performance depends on previous performance of a system because if it were better a few seconds ago, increased number of users could be accommodated and this would affect future performance. The mapping from previous outputs and control inputs to the current output is done through a function which may be linear or nonlinear. So, outputs can be estimated using techniques such as linear regression [52] for linear functions. In the case of nonlinear functions, a nonlinear model can be used. Control theory is a large field which has been used in electrical engineering, chemical engineering, mechatronics etc. So, A detailed discussion of control theory is out of the scope of this report. For a detailed discussion on control theory and feedback loop, readers may refer to [51].

2.4.5 Machine Learning

Machine learning can be classified into four categories which are *supervised*, *unsupervised*, *semi-supervised* and *reinforcement learning* [52].

1. Supervised Learning: Supervised learning learns from previous data or training data which are input and output pairs. From this data, it constructs a model of relationship between inputs and outputs.

2. Unsupervised Learning: In case of unsupervised learning, only previous inputs are provided and model needs to be constructed from these.

3. Semi-Supervised Learning: In semi-supervised learning, only for some small inputs, outputs are provided.

4. Reinforcement Learning: Reinforcement learning is the process of learning from interaction with the environment [53].

Machine learning problems can be classified into two parts namely *classification* and *regression* problems [52].

1. Classification Problem: Classification problems learn from the previous data and generates a qualitative response. Qualitative response can have value within a few classes, for example, performance can be within three classes high, low or medium.

2. Regression: Regression problems generate quantitative response which are numerical values. For example, performance can be represented by response time which can take any numerical values.

In self-adaptive system design, supervised learning and reinforcement learning have been used. For example, supervised learning has been used to predict a function which calculates metric values from *features*. Features are components that can be added or removed at runtime to achieve adaptation. As this is a regression problem, it can be solved with techniques like linear regression [52].

Reinforcement learning have been used by Kim et al. in [11] and Ho et al. in [54]. A reinforcement learning-based system always tries to choose the actions that maximize rewards. For this, it uses previous actions that produced high rewards and this is known as exploitation [53]. It also tries some previously unattempted actions to find a new way to achieve higher rewards, which is known as exploration [53]. These exploration and exploitation have to be balanced which is a difficult and vastly researched problem [53]. A reinforcement learning based system has four components such as *policy*, *reward function*, *value function* and *model of the environment* [53].

1. Policy: A policy is an association between different states of the environment and actions. In many cases, several policies are maintained in a lookup table where states are mapped to corresponding actions.

2. Reward Function: This is a mapping between a state and a number that indicates how much the system conforms to its goal. For example, a self-adaptive system may choose a reward function that returns how much its performance matches the expected one for a specific permutation of components.

3. Value Function: Reward functions show how much the system conforms to its goal for the current state or the next immediate state. However, value function corresponds to future rewards. Sometimes, value functions can be used instead of reward functions because a current high reward may be followed by multiple low rewards which can be calculated using value functions.

A reinforcement learning problem can be defined by *Markov Decision Process* containing set of *states*, *actions*, *transition probability* and *reward* which can be represented as a four tuple (S_m, A_m, P_m, R_m) [54]. S_m is a set of states which are specific configurations of the system. A_m is a set of actions, defined as any activity a system can perform. An action leads to change of the current state. P_m is the transition probability that the action a_m will lead to another state $s_m{}'$ from the current state s_m. R_m is the reward that is received after state transition occurs.

The techniques for reinforcement learning problems can be classified into *model-free* and *model-based* approach [54]. Model-based approaches combine estimated reward function and transition probability to produce a value function. In a model-free approach, policies are derived by learning value function from immediate rewards. A popular model-free algorithm is *Q-learning* [53]. In Q-learning, a random state is chosen and the reachable states from it are considered. The reachable state with maximum *Q-value* is chosen and its Q-value is calculated using Equation 2.1.

$$Q(s_n, a_n) = Q(s_n, a_n) + \alpha[R_{n+1} - Q(s_n, a_n) + \gamma \max Q_n(s_{n+1}, a)] \tag{2.1}$$

Here α is a constant and γ is the learning rate constant. $Q(s_n, a_n)$ denotes Q value for state s_n and action a_n. R_{n+1} indicates reward received after moving to the next reachable state.

At first all Q values are zero. These are continuously updated and stored, and these stored Q values are repeatedly used in the Equation 2.1 to derive Q values for next steps. The algorithm converges when all the Q values for all reachable paths have been found. When a system needs to choose a policy, it uses all these Q values for selecting the optimal path.

2.4.6 Software Product Line and Variability

As mentioned by Software Engineering Institute (SEI), *Software Product Line (SPL)* is a *"set of software-intensive systems"* which have some common features and have been developed from some core components [55]. SPL is the key to systematic reuse [55]. For this reason, concepts from SPL have been used to design and develop reusable self-adaptive systems [34, 7]. The core concern in SPL is to separate the products in a product line from three aspects which are *commonality*, *reusable variations* and *product specifics* [34].

1. Commonality: Artifacts that are common to all the products which need to be managed.

2. Reusable Variations: Components that are common to some, but not all, products that need to be managed.

3. Product Specifics: Every software has features that are very specific to the problem it addresses. So, after building the common parts of the software with the product line, these features are developed separately. These features that are specific to only one product, are not added to the product line.

It is noticeable from the three parts that product line design leads to commonality and variability management. These two parts are also known as *Domain Engineer-*

ing (DE) and *Application Engineering (AE)* respectively [34]. While DE aims to develop a set of common features, AE aims to derive a specific product from these features. In both of these stages, *variability* needs to be considered [34] because exploiting commonality demands filtering out common behaviors from variations. However, as mentioned by Abbas et al., to facilitate reuse in a self-adaptive system, three types of variability need to be considered which are *domain*, *cross-domain* and *runtime variability* [7].

1. Domain Variability: This refers to the differences among products within a product line. For self-adaptive software, two parts namely managing system (adaptation logic) and managed system (business logic) leads to two different product lines. In each product line, domain variability within it needs to be considered.

2. Cross-Domain Variability: This indicates variability across different product lines. As mentioned previously, self-adaptive systems have two separate product lines for managing and managed systems. Product lines in these two systems can be combined to create a multi-product line, which addresses this cross-domain variability.

3. Runtime Variability: Self-adaptive systems need to respond to changes that occur at runtime. This is why runtime variability has to be considered at the time of self-adaptive system design. The runtime variants need to be foreseen and added to the product line beforehand.

Most works on self-adaptive system seem to focus on runtime variability. In runtime variability, variants need to be added at runtime. For this reason, designing a SPL for self-adaptive system demands for analyzing the range of variations that may occur at runtime and adding the variants in the product line. This is also known as *Dynamic Software Product Line (DSPL)* [34]. DSPL is an emerging research area for designing self-adaptive system [34, 35]. However, Hallsteinsen

Table 2.1: Support of Design Patters in Four Functions of MAPE-K Loop

Patterns / MAPE-K Functions	**Monitor**	**Analyze**	**Plan**	**Execute**
Observer	**Yes**	No	No	No
Command	**Yes**	No	No	No
Chain of Responsibility	**Yes**	**Yes**	**Yes**	**Yes**
Composite	**Yes**	No	No	No
Bridge	No	No	No	**Yes**
State	No	No	No	**Yes**
Iterator	No	No	No	**Yes**
Proxy	No	No	No	**Yes**
Strategy	No	No	**Yes**	**Yes**
Decorator	No	**Yes**	No	No

et al. suggested that a mixture of DSPL and SPL may lead to better adaptation capability for a system [34]. In this case, there are two types of binding of variants which are *design time* and *execution time* [34]. At design time, DE and AE are followed to develop a product. Some variants which are related to static properties of the environment are also added at design time. At runtime, variants that are related to dynamic properties of the environment are bound. However, it is impossible to predict all the variability of the environment and include all the variants as core assets [34, 7]. So, DSPL also needs to be updated when such changes occur at runtime. This runtime evolution of DSPL can lead to better systematic reuse in self-adaptive systems. However, dynamic evolution of DSPL while ensuring correctness still remains a research problem [34].

2.4.7 Design Patterns

Design patterns are way of designing solution for a problem that occurs over and over again. Gamma et al. defined design patterns as *descriptions of communicating objects and classes* which have the aim to solve a general design problem from a particular point of view [56]. Gamma et al. introduced 23 design pattern for Object Oriented Systems which are popularly known as *Gang of Four (GOF) design patterns* [56]. To achieve a reusable self-adaptive system, the GOF design

patterns were applied to each of the functions of the MAPE-K loop [37, 36]. Salehie et al. mentioned that *Proxy* and *Strategy* patterns can be applied to effectors for a reusable design [6]. M.L. Berkane et al mentioned GOF patterns as *Technical patterns* [37] and applied these to all the four functions of the MAPE-K loop. Table 2.1 shows the GOF design patterns and their applications in the four functions of MAPE-K loop, as seen from the literature. Each of these, along with their role in self-adaptive system design, are briefly described below.

1. Observer: *Observer pattern* works like a *"publish-subscribe"* mechanism [56]. Observers register to subjects and any state change of the subject triggers a notification to the observers. This behavior can be used for publishing context information observed by Monitor to Plan function [37].

2. Command: *Command pattern* turns each command into an object and helps to hide the receiver from the invoker of the command [56]. The invoker of the command only executes a function in the provided command. The receiver is placed into the command by a client beforehand. The command executes and result is directed to the receiver. As the command class itself holds reference to the receiver, the invoker is effectively separated from this concern. In self-adaptive systems, Monitor function may use sensors to collect context information [37]. These sensors can be added or removed with the help of a *sensor factory* [36]. Here, Command pattern is used to separate sensor factory from Sensors to achieve better reusability [37].

3. Chain of Responsibility: In this pattern, a client passes request to a *"chain of handlers"* [56] but does not know who is going to handle that request. The request passes through the chain of handler, from successor to successor, until it finds an appropriate handler. This pattern is applied when a request is needed to be issued but the receiver of the request is known at runtime. *Chain of Responsibility* can be applied in all the functions of the MAPE-K

loop [37]. Passing the monitored data to Plan and then triggering adaptation are a chained sequence of activities, where this pattern is generally applied.

4. Composite: In *Composite pattern*, objects can be composed into a hierarchical form [56]. *Components*, which generally carry common behaviors of all the classes, are divided into *composites* and *leaves* [56]. Composite classes are composed of multiple leaves or combination of leaves and other composites. A leaf is a primitive object which is a specific class defining a particular behavior. Structuring classes in this form of hierarchy makes it easy for a client to use these because he only uses a component, without knowing whether it is a leaf or a composite. This pattern can be used in the Monitor function to acknowledge two types of sensors which are simple and complex [36, 37].

5. Bridge: This pattern is used when abstraction and implementer needs to be varied independently [56]. In *Bridge pattern*, abstraction carries a reference for the implementer interface. Abstraction is specified into multiple classes which implements the abstract methods. Implementer is also specified into multiple classes where these implement a separate method abstracted by the implementer. Thus, any specification of abstraction can invoke methods from any specification of implementer with the reference. This is used in the Executor function where reconfiguration rules and reconfiguration plans have different variants each but these plans may invoke different rules. Here, reconfiguration rules take the role of abstraction while reconfiguration plans take the role of implementer [37].

6. State: *State pattern* triggers a change in the behavior of an object as soon as its internal state changes [56]. Generally, the object is called context and holds a reference of a state class. As soon as client indicates a state change in the context, the reference is updated by the current state class. In self-

adaptive systems, this pattern helps to manage the different internal states (enabled or disabled) of components [36].

7. Iterator: This provides traversing the contents of an *"Aggregate Object"* (Collection or Map) [56] without exposing its internals. *Iterator pattern* is used to traverse through adaptation plans that may be buffered in the Execute phase, while an adaptation is taking place [37].

8. Proxy: *Proxy pattern* introduces a *"placeholder"* [56] for an object that can be used until the object is first referenced. The proxy object holds a reference to the original object. As soon as the object is referenced for use, the proxy object delegates the call to the original object. This pattern is used in the Executor function to define a placeholder for a component when it is enabled. The original component is used when it is referred for the first time. This helps to reduce the memory consumption during adaptation [6].

9. Strategy: This pattern helps to enclose a *group of algorithms* [56] under an abstraction which can be swapped independently from the client. In self-adaptive systems, *Strategy pattern* is used in the Executor function to support adding, removing and swapping algorithm for components. This pattern is also used to encapsulate adaptation strategies in the Plan phase [37, 6].

10. Decorator: This pattern is used when additional behaviors need to be supplement at runtime [56]. *Decorators* and the component that need dynamic addition of behaviors, both are abstracted by a common abstraction. Decorator holds a reference to the component and implement the abstract method from the abstraction. In this method, it adds the extra behavior and delegate the call to the component itself. In the Plan function, this pattern is used to dynamically compare the monitored values to thresholds for capturing goal violations [37].

Apart from these, another design pattern that is widely used is *Factory Method Pattern* [56]. Although this is not directly related to the MAPE-K loop, it is used in any phase to create any instance of an object. A class called creator class is defined where the abstraction of the factory method resides. However, this class delegates creation of specific objects to its subclasses. From the literature, it is evident that different design patterns, proposed for self-adaptive systems such as *Sensor Factory*, *Reflective Monitoring* [36] etc. are variants of the GOF patterns. Readers may refer to [56], where all the 23 GOF patterns have been discussed thoroughly.

All the discussed design patterns are able to achieve separation of concern. However, if concerns are scattered across all the classes, GOF design patterns cannot separate these. Separation of *cross cutting concerns* [57] are achieved through *Aspect Oriented Programming (AOP)*. Recent approaches in self-adaptive system design are considering AOP [57] [35]. This is because runtime variants are also cross cutting concerns as these variants may have been used in many places of the business logic code. Laddad mentioned that AOP is based on three concepts which are *joinpoints*, *pointcuts* and *advices* [57].

1. Joinpoint: A joinpoint is any point of the execution of a program. From the AOP perspective, a joinpoint is a point in a program where an additional behavior can be attached.

2. Pointcut: A pointcut is used to nominate a joinpoint, that is, a pointcut helps to identify joinpoints. For example, a pointcut may specify the pattern of a method call (joinpoint) where behavior needs to be added.

3. Advice: The additional behavior or code that needs to be added is known as advice.

The additional behavior addition at certain joinpoint is known as *weaving* [57]. In case of self-adaptive systems, the additional behaviors or variants need to be

added at runtime. This runtime weaving is known as dynamic weaving [57] [15]. Although dynamic weaving has potential to be incorporated with self-adaptive systems, only few literature have been found to address this [15] [58].

2.5 Summary

Self-adaptive system is a widely researched topic. This is why it is becoming popular in areas such as sensor networks, transportation, social service-based systems etc. to build context aware software. The design of self-adaptive system has been approached from multiple domains such as machine learning, architectural design, component-based software engineering, control theory etc. In this report, the basic concepts on self-adaptive system, its development and design have been discussed. The application areas have been mentioned and approaches from different domains for developing self-adaptive software have been explored. This discussion can help to understand the existing works in self-adaptive system design.

Chapter 3

Literature Review of Self-Adaptive System Design

In this report, the existing self-adaptive system design approaches will be discussed. In the literature, numerous self-adaptive system designs have been proposed. Most of these aim for achieving effective adaptation [32, 10, 54, 59, 12, 11]. However, engineering approaches of self-adaptive software and quality concerns such as reuse, modularity etc. are recently receiving importance [7, 35, 36]. Based on the type of techniques followed by these, six types of self-adaptive system design approaches are visible from the literature which are listed below.

1. Architecture-Based
2. Component Model-Based
3. Control Theory-Based
4. Machine Learning-Based
5. Software Product Line-Based
6. Design Pattern-Based

It has been observed from the literature that the third and fourth one are mostly concerned with the effectiveness of adaptation. The last two are concerned with reuse and the first two often takes a mixed approach. It is mentionable that, this classification is not orthogonal. For example, some machine learning-based approaches such as FUSION uses architectural models which are also used by architecture-based approaches. In the remaining sections, all these approaches are discussed.

3.1 Architecture-Based Approaches

Architecture-based adaptation techniques are dependent on architectural models [38]. These models are analyzed to detect goal violation. Then, the models are updated according to the adaptation decision. This update is also pushed to the effectors [21] which perform reconfiguration in the managed system. Several architecture-based adaptation methodologies have been proposed of which *Rainbow* [9], *MADAM* [33] and *Transformer* [60] are the most predominant ones.

3.1.1 Rainbow

Garlan et al. proposed the Rainbow framework [9] based on architectural model and style [38]. The main goal of Rainbow was to achieve adaptivity while ensuring reusability of the adaptation component [13]. Along with effective adaptation, reusability is also important because a reusable adaptation component can reduce the effort of building a complex adaptive system. However, reusability demands for an external or separated adaptation component. This component needs a complete model of the system because without it, this adaptation component cannot analyze and execute adaptation decisions.

Although an architectural model can fulfil the need for a complete model of the system, a number of issues arise regarding this. Firstly, different systems have

different architectures and so, different models. This variability must be considered to achieve reusability. Secondly, tailoring the adaptation component to specific system can be costly because different strategies and effectors need to added. A reusable adaptation component can reduce this cost. For this reason, adaptation component should be designed in such a way that reuse can be maximized.

In Rainbow, the whole adaptation infrastructure (adaptation component and the system) was divided into three parts which were *System-layer*, *Architecture-layer* and *Translation infrastructure* [9]. System layer infrastructure was related to the system itself and consisted of *probe*, *effector* and a *resource discovery mechanism* [9]. Probes collected data from the system, effectors executed adaptation decisions and resource discovery checked if a component was available before enabling or disabling it. Architecture-layer infrastructure consisted of *model manager*, *gauge*, *constraint evaluator*, *adaptation engine* and *adaptation executor* [9]. Model manager provided access to architectural model. Gauges worked like monitors and constraint evaluator analyzed the monitored value for goal violations. Adaptation engine carried the strategies and determined the actions needed to be taken for adaptation. Adaptation executor modified the model and passed the execution decision to translation infrastructure. This infrastructure carried mapping from architecture model to specific commands and elements in the system. So, the decision was turned into system specific entities and passed to appropriate effectors. Rainbow also used architectural styles to exploit commonality of various system architectures. Systems with same architectural style could reuse elements such as rules, strategies, parameters etc. among these to achieve better reusability.

Garlan et al. performed case study on a client-server application and a video-conferencing system. Effectiveness was evaluated based on performance (latency) and it was seen that Rainbow performed better that a system without adaptation. However, reusability was evaluated based on lines of code and Garlan et al. mentioned that a better evaluation was required.

Although Rainbow seem to achieve effective adaptation and reusability of the infrastructure, most of the elements such as strategies, rules, effectors etc. were different for two different architectural styles. Besides, strategies were hardwired into the system which made reusability a difficult task. It would have been better if specific strategies could be plugged in or out dynamically. It is also noticeable that Rainbow did not include any conflict resolution mechanism for adaptation decision. For example, if a system decides to add an extra server to improve performance, cost is also increased. These conflicting scenarios can harm the quality of adaptation. However, Rainbow exposed an important observation for self-adaptive system design, which was - combination of commonality and variability leads to better reuse. It is also mentionable that Garlan et al. observed adaptation to be effective after a few rounds from the adaptation decision. That means adaptation itself takes time and resources to execute. So, some sort of resource prediction mechanism can lead to better and stable adaptation. However, Rainbow was extended with resource prediction in [61]. Recently, Rainbow has been further extended to solve the problem of static strategies in [62]. Even in this solution, reusability is also an issue because these strategies are system specific and cannot be reused in another system.

3.1.2 MADAM

Floch et al. proposed the MADAM framework, specifically for adaptation in mobile devices [33]. In case of mobile devices, adaptation is difficult because of resource (for example, memory) constraint. Besides, assuring adaptation quality is also necessary. However, high quality adaptation needs rigorous analysis and selection of correct variants, which is costly in terms of resource [33]. This conflicting scenario makes adaptation nontrivial.

In previous approaches such as Rainbow, architecture model was a design-time artifact. However, as architectures can change at runtime due to adaptation,

for example, through an online update of new component deployment, only design time architectural model is not sufficient. Besides, static strategies do not fit properly in this context because static strategies cannot represent runtime variation in resource.

MADAM was similar to Rainbow from the perspective that it also used external adaptation. However, it used a runtime architectural model along with a design time model. These models were *component framework* [33] type where architecture was represented as a set of components. These component could also be composite, that is, could be composed of other components. The component framework model helped to make reconfiguration easy. For example, assume that a component is composed of some variants which may be used alternatively. Component framework model helped to attach one of these through a simple component addition operation to the model. Floch et al. also introduced *component parametrization* [33] where components contained properties representing what these offered (to user) and what these demanded (resource). A *property predictor function* [33] calculated associated utility of a property for a given context. The notion of *utility* [12, 33] helped to replace static strategies. In the operational state, as soon as a context change was detected, the runtime model was analyzed to find variants with highest utilities. These were selected and applied to the system through a reconfiguration. MADAM reconstructed the runtime architecture at initial deployment and after every reconfiguration to keep the model and the code synced.

Two case studies were conducted using MADAM on a janitor inspection system and a videoconferencing system. They showed the effectiveness of their approach in a way similar to Rainbow. They also mentioned that their approach was reusable because of external adaptation, runtime model and replacement of static strategies. They also conducted a pilot study in industries with MADAM which helped to validate their approach.

The pilot study mentioned in the previous paragraph also showed that defining utility functions is a difficult task for an architect. Besides, analyzing utility for a large combination of variants also poses a problem. Thus, MADAM helps to improve reusability but reduces simplicity and scalability. However, it would have been better if utility values could be derived. It would also be useful if the context information, rather than utilities, could be used to find the best component, because context information is always available through monitor.

3.1.3 Transformer

Ning Gui et al. proposed the Transformer framework for increasing reusability and supporting conflict detection and resolution [60]. Adaptation strategies are system specific. Besides, all the adaptation strategies are written in a single monolithic module which makes reuse of these very difficult. However, dividing the strategies into different independent modules also poses problem because different strategies may suggest conflicting solutions. Thus, achieving conflict resolution and reusability together is difficult.

It is challenging to compose multiple strategies at runtime to take a single adaptation decision. This is because finding the most appropriate strategies for an adaptation becomes difficult. This type of composition may also lead to conflicts. It is difficult to detect and resolve these conflicts because strategies are into multiple independent modules. Another challenge is to modularize strategies in such a way that reusability is improved.

Transformer separated the strategies based on goals. Strategies of a particular goal were composed in a module which they termed as *Composable Adaptation Planner (CAP)* [60]. Each CAP had a *context preference* [60] which represented the preferable environment to use the CAP. Context preferences were represented as a set of three tuples *(context factor, preferable value, impact factor)* [60], where context factor was the context name, preferable value was the preferred value of

the context and impact factor indicated the impact of the context on that CAP. When a context change was detected, each CAP was matched with the context based on the current context value and preferable value, weighted by impact factor. The CAP with matching value or *Context Matching Degree (CMD)* [60] greater than a threshold was selected. However, if multiple CAPs were selected, a *model fusion* [60] element resolved the conflict. Model fusion chose the CAP with highest CMD if the strategies led to a component addition or removal. If the strategies led to update of a value, either all the suggested values from the CAPs were averaged or the highest CMD approach was followed. Finally, the adaptation plans were executed by effectors.

Transformer was applied on a mobile videoconferencing system. Multiple CAPs were deployed based on two conflicting goals which were quality of video and battery level. It was seen that adaptation was effective when Transformer framework was applied. However, the reusability of the framework was qualitatively analyzed by comparing it with other frameworks. No quantitative result was provided for reusability.

Although this framework provides conflict resolution and strategy reuse, the strategies are still static entities. Besides, deciding the context preference values and impact factors is as challenging as deciding utility values. It is also noticeable that goals may be dependent on one another in a way that one cannot be satisfied without satisfying the other. Neither of the mentioned frameworks address this issue. However, one of the most important observations from Transformer is that, separating strategies based on goals and composing these according to the context at runtime can improve reusability.

3.1.4 Summary of Architecture-Based Approaches

On the previous sections, architecture-based approaches for self-adaptive systems which are Rainbow, MADAM and Transformer have been discussed. Apart from

these, there are also some other architecture-based methodologies which extend these frameworks. The resource constraint problem was solved by a Rainbow variant in [61], static strategy problem was solved in [62] and a language called *Stitch* for writing strategies in Rainbow was discussed in [63]. Based on MADAM, another framework named MUSIC [64] was proposed. In all cases, architecture-based adaptation achieved some reusability because of explicit architectural model and external adaptation. However, removing static strategies while ensuring simplicity is challenging in all the proposed architecture-based approaches.

3.2 Component Model-Based Approaches

Component Model-based approaches relies on the managed system following a specific component model [49]. The component model also need to be dynamic and reflective for reconfiguration [41]. Using a specific component model makes system monitoring easier and reconfiguration straightforward. Several component model-based approaches have been proposed. *Fractal component model-based approach* [13], *Fractal and dynamic AOP-based integrated approach* [15] and the *K-Component* framework [41] are the most discussed ones in the literature.

3.2.1 The K-Component Framework

The K-Component framework [41] was one of the earliest approaches for component-model based self-adaptive system design. In self-adaptive system, reconfiguration of components, maintaining the integrity of the system is difficult because reconfiguration analysis needs a structured representation of components. Besides, separating the adaptation code from the business logic code is also important for achieving reusability. Structured components and separation of concerns both can lead to reusable and modular self-adaptive software systems.

In a self-adaptive system, adaptations must not lead to a state where the

system becomes unstable. It may happen when adaptation is performing reconfiguration on a code which the business logic is executing (that is, concurrent access). For this, reconfiguration protocols are needed. Specifying and maintaining these protocols is nontrivial because it requires system to be structured in a way so that it can be easily accessed. Separation of concern is also difficult because of dependencies between adaptation and business logic in different stages such as during reconfiguration.

K-Component framework aimed to develop self-adaptive system applying both architecture and component models. The meta architecture model of the system was described as a graph where nodes were components and their interfaces, and edges were interactions between components. Adaptation code was written in a module called *adaptation contract* [41] which was written in a different language. Adaptation contracts held condition-action rules which were triggered based on events. These rules updated the graph when events indicated goal violations. The component model of the managed system was developed with *Interface Definition Language (IDL-3)* [65] from a famous middleware named *CORBA* [65]. The component model allowed structured design of components and helped to the graph updates to the corresponding components. A structured component model also helped generating the architecture graph from static analysis of code.

This framework was a conceptual model and was not tested. Although K-Component framework uses a component model for reconfiguration, it uses static condition-action rules. Besides, generation of graph from static code may be erroneous if any code fragment is dynamically inserted (dynamic binding [56]). Most importantly, building a system to a specific component model makes reuse within the same component model easy but between different component model costly. This is because the full code base need to be refactored to that specific component model. It would be useful if the amount of refactoring could be minimized.

3.2.2 Fractal-Based Framework

David et al. proposed a Fractal-based framework [13] where the goal was to achieve complete separation of concerns. Reusability is best achieved when two modules can be separately developed and deployed independently. In case of self-adaptive system, it is desirable that the business logic is developed separately from the adaptation logic. At the time of deployment, the business logic can be augmented with the adaptation logic to produce a complete self-adaptive system. However, the interaction between these two parts throughout the lifetime of the system makes complete separation a very difficult issue.

For achieving complete separation of concern, the system must be designed in a structured way. The reason is unstructured systems lead to uncertainties during reconfiguration and monitoring. However, it is also necessary that the structured representations support reflection which is the capability of a system to analyze itself at runtime [13]. Besides, the system should be adaptable [13, 41], that is, it should have built in reconfiguration mechanism which is also challenging. Online deployment for supporting unpredictable scenario also poses a challenge because most of the component models do not support this.

This framework divided the whole self-adaptive software into three parts which are *Fractal component model*, *Context-awareness service* and *Adaptation policies* [13]. The managed system was developed or refactored following Fractal component model. This component model was also customized to support online deployment by extending the controller and enabling interception. Context-awareness service provided current context information. This was divided into three stages namely *acquisition*, *representation* and *reasoning* [13]. Acquisition is similar to monitoring with probes in Rainbow, which collects raw context information. Representation structures the information tagging it with corresponding resource. Reasoning helped to compose synthetic attributes by combining measures and kept these updated. The context information from context-awareness service was

used by adaptation policies to query for goal violations. A goal violation triggered reconfiguration led by these policies which included component addition or removal according to Fractal.

The framework was tested on a small image browser. Scenarios were considered around the decision to enable or disable a cache. It was seen that adaptation was effective and core system could be augmented with adaptation logic. However, no quantified evaluation was provided.

An interesting observation from this framework is that, under assumptions that managed system is adaptable, component models lead to maximum reuse. However, component models may lead to the problems described previously. Apart from this, what specifically defines an adaptable system and how to incorporate such adaptability with Fractal was not mentioned. Thus, this framework is reusable under specific assumptions, but these assumptions themselves lead to reusability problems.

3.2.3 Fractal and Dynamic AOP-Based Approach

Yuankai Wu et al. proposed an integrated approach with Fractal and dynamic AOP [15]. As mentioned previously, component model lead to better reuse but full code base refactoring may be needed. However, this is costly and time-consuming. Moreover, complete refactoring becomes impossible in case of applications that have large code bases. The conflicting requirement of following a component model and refactoring less codes poses a major challenge.

Following a component model and reducing refactoring are completely opposite requirements as mentioned in the previous paragraph. However, crosscutting concerns further complicates scenario. These are not supported by any component model and no amount of refactoring can remove these. Thus, addressing crosscutting concerns while following a specific component model requires attention.

This approach proposed a simple solution to this problem. In this technique,

the full application was structured with Fractal except the crosscutting concerns. These were placed inside the code following a traditional AOP-based approach. Aspect weaving helped to enable or disable these at runtime. While Fractal was used for component reconfiguration, AOP was used for aspect reconfiguration. Thus, the integrated model solved the crosscutting concern problem and reduced refactoring for these concerns.

A case study was performed on a Public Service system for Self-taught Examination (PSSE). Scenarios covering aspect weaving and Fractal-based approach were considered. In all the cases, adaptation was achieved. However, no quantified data and evaluation was provided on effectiveness or reusability.

Although this technique solves the crosscutting concern problem and seems to reduce refactoring, it is actually not significantly reduced. Often applications carry only a small number of crosscutting concerns. So, the amount of codes needed to be refactored remains almost unchanged. This indicates that this problem is challenging to address. However, Identifying interactions between adaptation and business logic and structuring only those interacting parts using a component model may be useful.

3.2.4 Summary of Component Model-Based Approaches

Most of the component model-based approaches discussed here aim to build a reusable self-adaptive system. However, these require following a specific component model all over the code base. So, all of these suffer from the problem of refactoring large code bases which is impractical due to limited time. It is also mentionable that, a variant of Fractal was also proposed by David et al. based on AOP for self-adaptive system [58]. It was also seen from the literature that this variant also led to similar problems.

3.3 Control-Theory Based Approaches

Control theory has been applied successfully in physical systems and different subjects such as mechatronics, electrical engineering etc [51]. Researchers also attempted to use control theory for self-adaptive system design because control theory is mathematically well-grounded [10]. *Hierarchical model-based autonomic control* by Litoiu et al. [59], feedback control for *Model Reference Adaptive System (MRAS)* by Shaw et al. [32] and *control design process* mentioned by Filieri et al. [10] are the most prominent approaches from the literature.

3.3.1 Hierarchical Model-Based Autonomic Control

This model proposed by Litoiu et al. was one of the earliest approaches for self-adaptive software design based on control theory. One of the main goals of self-adaptive systems is to satisfy quality of service requirements [59]. Different control algorithms such as *threshold-based control*, *policy-based control* etc. [59] exist for adaptation. However, none seem to cover all types of situation that may occur during operational stage. So, achieving quality of service goals in varieties of system is difficult. Besides, maintaining high service level is also important but only control theory cannot achieve this. This is because it requires component reconfiguration and so, update of the managed system model. Control theory does not support such runtime model change during its execution.

A system can be composed of multiple components where complex goal violations may occur. In a traditional control-based system, only one component is controlled and component interactions are ignored. It harms service level and quality of service. However, designing such multicomponent interaction-based model is challenging because it requires augmenting the complex control loop with several models. Supporting model update with the control loop is difficult because control theory does not support this, as mentioned previously.

The hierarchical model proposed by Litoiu et al. divided the whole system into a managed component and three levels of controllers attached to it. The managed component was the core system and consisted of application code, sensors and effectors. Controllers were structured into three levels namely *component controller*, *application controller* and *provisioning controller* [59]. Component controllers kept a model of a component and inspected it for goal violation. The model consisted of previous inspection and action information. In case of a goal violation, the model was analyzed to predict future adjustments of the controlled system parameters based on control theory-based methods. If this failed to adapt the system, the application controller tried to analyze a component interaction model in a similar approach to tune system-wide parameters, such as number of load-balancers. If this also failed, the provisioning controller was used to load another alternative component through reconfiguration. This method also allowed to use different control algorithms such as threshold-based and policy-based approaches to build the *performance model* [59]. As control theory supported incorporating time with input and output of the system for prediction, time requirement could also be attached to achieve timely adaptation.

Litoiu et al. did not provide a quantitative or qualitative evaluation of their technique. However, their approach indicated that control theory-based methods were useful and could be incorporated with system models to meet quality of service goals. As different controllers and system were separated, some reusability was also achieved. However, as the controllers contained the models within themselves and were very tightly coupled, the achieved reusability level was low.

3.3.2 Feedback Control for MRAS

Shaw proposed a feedback control for MRAS where a reference model is kept for analysis [32]. The model proposed by Litoiu et al. was also for MRAS but it did not consider visibility of control [32] explicitly. An explicit or visible control loop

is important for reusability because it enables separation of concern between a controller and a system. However, it is also important to know how the control loop will be realized. It means that all the requirement specification, design and implementation steps for control loop must be explicit, that is, performed visibly in a systematic way.

Specifying a feedback control for MRAS along with the steps mentioned previously is challenging because control theory is generally suited for physical entities. Control loop design and development leads to questions such as how goals should be modeled, what should be separated and how to implement controller and system in a modular way. As discussed previously, all these questions are difficult to answer.

The feedback loop by Shaw was composed of a feedback controller and a model of the system, which the controller used to choose best configuration parameters for adaptation. From this perspective this model was similar to the one by Litoiu et al. except that, this model used only one controller. The main contribution of this technique was to develop a complete software engineering process for feedback control-based system. They mentioned four steps for software engineering of control-based system which are *requirement specification*, *design*, *development* and *testing (verification and validation)* [32]. In the requirement specification stage, goals were identified and quantified, along with time and resource constraints. In the design phase, the components of the controller and the system were separately identified and an adaptation strategy (for example, threshold or policy-based) was chosen. The feedback control-based system was developed in the implementation stage where controller was not a completely separate entity in the code. In the final step called verification and validation, the system was checked for appropriate error calculation, time management, stability of adaptation etc.

This methodology was not validated with any case study or quantitative evaluation. Nevertheless, the approach by Shaw showed the importance of making

the control loop a *first class entity* [32] for effective and reusable system by following explicit software engineering process for it. However, it mentioned that codes of control and core system can be intermingled. This contradicts in the way to achieve a reusable controller. Besides, the mentioned four steps are very conceptual and does not exploit the established mathematical background of the control theory.

3.3.3 Control Design Process

A formal control-based system design approach based on mathematical support of control theory was proposed by Filieri et al [10]. As discussed previously, control theory-based approaches did not exploit the mathematical properties of control theory to develop a complete design process. Representing a controller mathematically provides formal foundation for its effectiveness. However, a complete process for control design is also important along with its mathematical representation in order to make the control explicit.

Apart from complexities regarding development of a control design process, establishing a mathematical control theory model for self-adaptive system is difficult. Software are not precisely measurable as physical entities [10]. As control theory is based on accurate measurement of system goals, it poses challenges. Besides, stable and timely adaptation requires time to be incorporated into the model. However, time is varying and considering it makes the equations differential ones, which is complex to analyze. How goals, monitored information and effectors or *knobs* [10] can be mathematically modeled also introduces additional complexity.

In this model, Filieri et al. divided the control design process into six steps namely *identify the goals*, *identify the knobs*, *devise the model*, *design the controller*, *implement and integrate the controller*, and *test and validate the system* [10]. Goals needed to be quantifiable and measurable. They identified three types

of goals namely *setpoint*, *range* and *minimization or maximization* [10]. Setpoint goal used a reference value to track goal violations. For example, a system may fix a goal with response time less than 2 nanoseconds. Range goal used number range instead of fixed value. For example, a system may set a goal with response time between 2 to 5 nanoseconds. Minimization or Maximization goal aimed to minimize or maximize any value respectively. For example, minimum response time or maximum performance can be the goal of a system. The next step was to identify the knobs that were system configuration parameters that could be updated. Then, the model was devised where the mathematical relationships between the knobs and the goals were established. Mathematical equations were formed involving input variables, state variables and output variables [10]. Another technique to form the relationship was to derive a *transfer function* [10]. Transfer functions were Z-Transformed ratio of input and output functions with time requirement where the input and output functions were derived from historical data. *Z transform* [51] was used to convert values from time domain to frequency domain for removing differential equations. Thus, the result of this step was a mathematical model which could predict the next knob choice, given current knob choice. After devising the model, the full controller was designed. To do this, manual approach could be followed with trial and error to find the best design. However, an analytical model was proposed which took the previous transfer function of the core system and integrated it with a transfer function for the controller. This integrated transfer function was converted to time domain by reverse Z-transform which produced an equation to predict the next output of the system, given current knob and error value. This model was implemented using tools such as *Simulink* [66]. Finally, verification and validation of the system was accomplished where different methods such as finite automata-based or statistical distribution-based [10] approaches could be followed.

A video encoder system was used to demonstrate the different steps of this

methodology. However, a complete case study was not provided. Quantitative evaluation to show how this approach improved controller design was not provided. The contribution of this paper was to establish the fact that self-adaptive systems based on control theories can achieve very sophisticated and effective adaptation. Although the control design process helped to understand the usefulness of control theories in self-adaptive system design, it was still difficult to reuse the controller. This is because it was dependent to knobs and core system model. It was also not discussed how model updates could be handled.

3.3.4 Summary of Control Theory-based Approaches

Although control theory-based self-adaptive systems provide formal assurance [10] to their effectiveness, complete reuse can be a complex issue. The approaches discussed here do not seem to address this issue. Apart from these techniques, varieties of feedback control based systems were also discussed in [5] by Brun et al. They mentioned that a reusable code base for these self-adaptive systems was much desired. So, it is evident that control theory-based self-adaptive systems can be very effective if reusability can be ensured.

3.4 Machine Learning-Based Approaches

Machine learning based approaches use the effect of previous reconfigurations on goals to find out better reconfiguration approach for goal conformance. Most of the machine learning based techniques focus on reinforcement learning [11, 54]. However, Esfahani et al. proposed a framework which used supervised learning for utility function derivation [16, 12]. In the subsequent sections, *Q-learning based method* by Kim et al. [11], *model-based reinforcement learning technique for self-adaptive system* by Ho et al [54] and *FUSION framework* [16, 12] by Esfahani et al. will be discussed.

3.4.1 Q-learning Based Method

A model-free Q-learning based approach was proposed by Kim et al. which aimed to improve planning method for a more dynamic self-adaptive system [11]. Most of the approaches discussed previously used offline planning which are static condition-action rules. The problem of offline planning is that these rules are hard to update and it assumes that rules are known in design time. However, as mentioned by Kim et al., these rules are rarely known in design time and thus offline adaptation is often not effective [11]. So, an online approach is needed where best rules for a condition can be derived dynamically.

For online planning, goals of the system must be known. The goals also must be quantified to choose the best plan. However, choosing and quantifying goals in a systematic way is challenging because goals are often abstract. As mentioned in the previous paragraph, only the best rules need to be chosen according to the context. This demands for a method where rules can be provided with a feedback, and rules with highest positive feedback will be chosen. Mathematically modeling this mechanism is also a problem.

Kim et al. used *states* and *actions* to define the planning [11, 54]. States were represented by a *situation*, *state type*, *range* and *architectural model* of the system [11]. For example, if a system has a performance related goal, sudden low performance can be a situation. Here, performance is the state type and its range can be high, medium or low. Kim et al. divided the online process in five parts namely *detection*, *planning*, *execution*, *evaluation* and *learning* phase [11]. In the detection phase, when a situation occurred, the corresponding state and its allowed values were passed to the planning phase. In planning, upon receiving the state information, a random number ϵ was assumed between 1 and 0. If it was less than a prespecified value, an action (that is, reconfiguration) was randomly chosen. Otherwise, the action which had the highest Q-value (constructed from learning phase) associated with this state was chosen. This action was executed

in the execution phase. In the evaluation phase, a reward function [53] calculated the reward of this action. This value was used in the learning phase to update the Q-value of the state-action pair with Q-learning [53].

This method was tested on *Robocode* [67], a robot battle simulator. A robot was chosen to test on and a very strong opponent called Antigravity 1.0 was selected. It was seen that after a few rounds of learning, the chosen robot could beat Antigravity 1.0 frequently. Kim et al. also tested the exploration-exploitation [53] issue related to all the reinforcement learning-based methods. They set ϵ to 0 which meant that only Q-values would be used, and so no exploration would take place. Using the learned Q-values from first experiment, they showed that the chosen robot could consistently beat Antigravity 1.0. They also set to 1 and 0.5 to show the effect of full exploration and exploration-exploitation respectively. It was seen that using exploration-exploitation helps in new situation to adapt and using only exploration leads to average results.

The Q-learning based method by Kim et al. introduced a way to improve the effectiveness of adaptation by choosing best actions up-to-date. However, as mentioned by Ho et al., Q-learning is extremely time-consuming if feature space is large [54]. Besides, after reconfiguration, the model is updated and so, the state space changes. So, the previous Q-values become invalid. For this reason, although this approach is effective for small scale self-adaptive systems, it does not fit properly for large scale systems and reuse becomes difficult.

3.4.2 Model-Based Reinforcement Learning Technique

Ho et al. proposed a model-based reinforcement learning technique [54] for self-adaptive system to solve the aforementioned slow learning problem. Model-free reinforcement learning was used in the literature [11] in order to achieve effective adaptivity but using an environment model was not considered. Using a model of the environment helps to learn how the context works. Then, this knowledge can

be used to take decisions faster, rather than just relying on continuous upgrade of reward values.

Although it is clear that a model of the environment can help to make learning faster, how the model will be constructed is a challenge. According to the markov decision process [54], transition probability represents the probability of a specific action leading to a specific state from another. Environment model can be represented by these transition probabilities because these represent rules of the environment [54]. In this case, another challenge of deriving these probabilities appear which is complicated due to uncertainties in the environment.

In this method, a *Bayesian approach* [54] was followed to construct the transition probabilities. To state formally, probability $P(s'|(s,a))$ was calculated for every state and action pair, where (s,a) represented current state-action pair and s' was the next state. For this, the number of times a specific state-action pair had occurred was calculated from previous information. This number was incremented continuously as new pairs were seen. This was used to calculate the transition probability with Bayesian approach. Thus, a model of the environment was available which could be continuously updated. This model was used to calculate a value function [53] which helped to choose an optimal action or policy.

Ho et al. conducted a case study on cloud servers. The goal was to adapt response time by adding or removing resource, or by lowering content quality. The case study showed that this model-based method converges fast and takes better decisions. However, they also performed another simulation by reducing training time. It was seen that this method took decisions almost as same as Q-learning based one but the adaptation was more stable.

Model-based methods provide a time efficient solution to adaptation. However, model-based methods are computationally expensive as large amount of state space need to be traversed. Besides, both of these reinforcement learning-based methods are very system specific by nature. So, these cannot ensure reusability

which is necessary to reduce development and maintenance effort. Thus, these two are effective self-adaptive system algorithms but not appropriate solution for a reusable self-adaptive software system.

3.4.3 FUSION Framework

Esfahani et al. proposed the FUSION framework [16, 12], a learning-based approach for self-adaptive system design. In the reinforcement learning-based methods, time and computational complexity was prevalent. Besides, none of these methods mentioned about usage of external adaptation which is essential for reuse. To provide a framework for self-adaptive system that is dynamic and reusable, but less complex, is difficult. This is because any dynamic reward-based models, as seen from the literature, lead to large state space [54, 11].

Handling computation complexity in a online learning-based system can be challenging because the full state space must be considered for learning accurately. However, variants, also called *features* [16, 12], may have dependencies among themselves. These features can also be restricted by constraints. These dependencies and constraints restrict the feature space and can be used to reduce complexity. However, handling these constraints and dependencies throughout the adaptation process is an intricate task. Besides, the validity of the learnt knowledge after reconfiguration need to be assured to.

FUSION consisted of two cycles named *learning and adaptation* [16, 12]. Learning discovered relationships between features and metrics. A feature is a variation point of the software. Metric is usually an equation that measures something, for example, response time is a metric for performance. Learning consisted of two activities that resulted in a feature-metric relationship function. The first one was called *observe*. Its main goal was to normalize the raw metrics and test the learned functions continuously. Metrics were normalized to make those comparable. Testing compared a predicted metric value from the learned functions with

actual observed values to find error. A wrong decision from the learned function represented a new pattern and the function needed to be reformed. The next step of learning was *induce* where a significance test was performed. The significance test determined features that had the most impact on each metric. This was necessary because it reduced the number of features to consider. Then, a well-known learning algorithm such as *linear regression* [52] or *M5 model tree* [68] was applied which resulted in a feature-metric relationship function. Next step was the adaptation cycle which consisted of three activities called *detect*, *plan* and *effect* [16, 12]. In detect step, utility functions were used. These utility functions resulted in zero when a metrics value exceeded an accepted limit and resulted in a positive value otherwise. The use of utility function made the detection process mathematically justified. In planning, shared features that affected common metrics, were figured out using the functions from the learning phase. As these affected the same metrics, these also affected the same goals. So, these were conflicting goals. The target of FUSION was to figure out a selection of shared features that maximized the utility of the system. This represented an optimization problem and could be solved by many known optimization techniques such as *linear optimization* [69]. In this way, FUSION modeled the plan step in a mathematical and implementable way. The final step was effect and it consisted of enabling or disabling a feature according to the optimized selection in the plan step. Thus, FUSION delivered a structured solution to self-adaptive system design which was implementable using the current technologies in hand.

Esfahani et al. tested their framework rigorously on an online travel reservation system. They selected the goals of quote response time, travel agent reliability, quote quality and accountability. For each experiment, they evaluated the technique on four environments which simulated similar or normal context, varying context, unexpected event with emerging pattern and unexpected event with no pattern. Accuracy of the learning and how performance changes with emerging

pattern was evaluated. The computation complexity was also tested by comparing it with a system that selected all the features for metric calculation. Finally, the goal conformance during the effect step was tested to see whether reconfiguration led to any constraint violation. In all these cases, FUSION performed better than any other methods in the literature. However, though it was mentioned that this framework embraced separation of concern, no quantitative evaluation was provided.

FUSION seems to solve the problems mentioned in previous sections. However, Esfahani et al. mentioned that the tool developed with this framework needed to be specialized to different systems for use. This indicates that FUSION, as a tool, was not reusable. They also did not mentioned how system specific feature, metric and utilities could be managed for a reusable adaptation logic. Separating the feature, metric and utilities from the system and inserting these at runtime may lead to a reusable solution.

3.4.4 Summary of Machine Learning-Based Approaches

It is understandable from the above discussion that machine learning-based approaches have the dynamism and flexibilities to be used for self-adaptive system design. However, research challenges still exist. The use of utility functions has been criticized in [70, 36] because it requires an optimization problem to be solved for every adaptation which is costly and it is application specific. Apart from this, machine learning-based approaches are strong candidates for developing effective and reusable self-adaptive systems.

3.5 Software Product Line-Based Approaches

Software product line (SPL) is an emerging area that aims to achieve systematic reuse [55]. The concepts of SPL has recently been found as useful in self-

adaptive domain. A few approaches have been proposed based on SPL in order to achieve reusable self-adaptive systems. In the next sections, *Dynamic Software Product Line-based approach* by Hallsteinsen et al. [34], an approach named *MODELS@RUN.TIME* by Morin et al. [35] and the *ASPLe* framework by Abbas et al. [7] will be discussed.

3.5.1 Dynamic Software Product Line-Based Approach

Hallsteinsen et al. showed that Dynamic Software Product Lines (DSPL) can be applied for reuse in self-adaptive systems [34]. A lot of self-adaptive system design frameworks opt for effectiveness, leaving out the concern for reuse, as seen from the previous sections. The complexity of developing the adaptation component demands for reuse which can help build such components by composing existing implementations.

Although reuse have been achieved for general purpose software with modularization, design patterns, SPL etc., self-adaptive system imposes a major difficulty regarding managing and managed systems coupling. By the definitions of self-adaptive systems [23, 12], adaptation components or managing systems are tightly coupled with managed system by nature. Thus, these are system-specific. However, adaptation components also contain modules that are common across projects. For example, comparing with threshold is a common algorithm in self-adaptive system. So, managing this variability and commonality to achieve systematic reuse is hard to accomplish.

Hallsteinsen et al. proposed a conceptual model to capture runtime variability and commonality in self-adaptive system [34]. They suggested using a DSPL to find and restrict variants that can be attached to variation points at runtime. The domain engineering phase of DSPL developed common variants for supporting reuse across different products. The application engineering phase specialized the variants to specific products. They mentioned that DSPL can be integrated with

SPL where SPL helped to build the system at design time and DSPL supplied variants for runtime [34]. However, they also indicated that the DSPL need to be updated at runtime, as all variants cannot be foreseen in the domain engineering phase. This runtime evolution of DSPL was mentioned to be a research challenge [34].

This model helped to indicate the fact that a DSPL-based approach is possible for both design-time and runtime reuse. However, this was only a conceptual model and so, specifics of the model was not discussed. A more detailed model may help to build a reusable DSPL-based self-adaptive system.

3.5.2 MODELS@RUN.TIME

This method was proposed by Morin et al. and it used DSPL as a core element of its design [35]. From the previous section, it is evident that DSPL-based approaches need more detailed design guidelines. More specifically, how the DSPL variants can be generated and modeled with existing approaches to build a complete self-adaptive system needs to be clarified.

Making DSPL a core part of self-adaptive systems leads to some difficulties. Firstly, it needs to be answered where the DSPL will be used and how it can be used. It is also important to specify whether a variant implementation or a variant model will be used. The first one has the facility of easy deployment but a rollback in case of an error may not be possible. The second one supports error checking but needs to be updated frequently.

This technique modeled DSPL as a feature model, with feature dependencies and constraints. It used a *complex event processor* [35] to catch runtime events and notify a *goal-based reasoning engine* [35]. This component chose the feature selection that best conformed to goal and derived a DSPL model based on this selection. Then, an *aspect weaver* [35, 57], which was used to weave components to the architectural model, updated the architectural model according to the new

DSPL. This model was passed to *configuration checker* [35] which checked for any inconsistencies in the model. Finally, the system was reconfigured based on the architectural model.

This approach was tested on a *DCRM* system developed by *CAS Software* [71]. They tested an event notification where users changed platform (PC to mobile) and adaptation logic needed to load appropriate logic and interface for that specific platform. It was seen that adaptation was performed. However, they did not show whether the adaptation technique chose the best configurations to address a scenario. They also did not quantitatively show how reuse was improved.

Although MODELS@RUN.TIME developed a complete adaptation mechanism, it restricted only runtime variability using DSPL. It did not mention how different components of adaptation logic or the full adaptation components can be reused across products. More specifically, they did not model design-time variability with SPL. However, integrating DSPL with SPL can lead to better reuse [34], as mentioned previously.

3.5.3 ASPLe Framework

This framework, proposed by Abbas et al., aimed for integrating SPL with DSPL [7]. Self-adaptive systems are different from other general purpose systems because these involve two entities, managed and managing systems that interact continuously. This is why the domain engineering and application engineering processes of SPL do not fit directly into the self-adaptive system domain. So, SPL needs to be specialized for self-adaptive systems while integrating it with DSPL.

For self-adaptive systems, two product lines exist for managing and managed system separately. However, these two product lines are not independent [7]. For example, monitoring needs probes to be present inside the managed system. This is why achieving complete reuse considering a single product line such as adaptation product line is impossible [7]. Horizontal reuse or reuse across multiple

products is also challenging for the same reason.

In the ASPLe framework, three steps instead of two in traditional SPL was mentioned namely *domain engineering*, *specialization* and *integration* [7]. In the domain engineering phase, common artifacts or modules were developed. Here, artifacts common to all the members within a domain were implemented fully. However, in case of artifacts which differ from domain to domain were implemented in an abstract way and *hooks* [7] to specialize these were provided. In the specialization stage, a domain analysis was performed to see if the domain of the system and the product line in concern were the same. For same domain, reuse was straightforward. For different domains, the domain gap was identified and the hooks were used to derive components specialized to the domain. Finally, the interaction interfaces between managing and managed system were implemented in the integration phase. The variants were also derived and implemented in the same way.

The ASPLe framework was not evaluated to show the effectiveness of the approach. It is notable that, though this framework combined the SPL and DSPL in a systematic way, it did not solve the runtime evolution problem of DSPL. Besides, Abbas et al. mentioned that interactions between managed and managing system at runtime is a major bottleneck for reuse but did not capture this in their framework.

3.5.4 Summary of Software Product Line-based approaches

As mentioned previously, SPLs to design self-adaptive system is a recent approach. So, much remains to be explored in this area. The dynamic evolution of DSPL, integrating SPL and DSPL with self-adaptive system life cycle model and using SPLs to derive a complete self-adaptive systems are some of the areas of interest [34]. However, recent SPL and DSPL based approaches do not integrate the product line with the self-adaptive system design methods. This is required to

develop effective and reusable self-adaptive systems. Nevertheless, as seen from SPL-based approaches to other general-purpose systems, proper use of SPL can lead to major increase in reuse [55].

3.6 Design Pattern-Based Approaches

Similar to software product line, design pattern is also a recent topic in self-adaptive software system. Although design patterns were implicitly used in different self-adaptive system designs, these were not catalogued until Ramirez et al. produced a list of twelve design patterns [36]. Later, Berkane et al proposed a design pattern-based approach [37], which also used software product line support to build reusable self-adaptive systems.

3.6.1 Design Pattern Catalogue for Self-Adaptive Systems

As numerous design methods for self-adaptive were proposed, it was much needed to introduce design patterns for self-adaptive system design. The reuse related to design pattern is different from product lines. Product lines enable reuse of product artifacts where design patterns provide solution to a recurring problem, thus enable reuse of a concept. However, design patterns also seem to take separation of concern as an important factor [56]. This is why these can be useful for developing reusable adaptation component.

As mentioned previously, a number of design patterns were used in different self-adaptive system design methodologies. However, as these were implicitly used, making these explicit is an intricate task. Producing a design pattern catalogue as the one by Gamma et al. [56] also imposed challenges because the components of self-adaptive system contain dynamic properties and constraints.

Ramirez et al. discussed the design patterns for self-adaptive system in three classes namely *monitoring pattern*, *decision-making pattern* and *reconfiguration*

pattern [36]. Monitoring pattern included *sensor-factory*, *reflective monitoring* and *content-based routing* [36]. Sensor factory used sensors for monitoring and these sensors could be simple and composite, consisting of multiple sensors. Reflective monitoring used reflection like Fractal [13]. Content-based routing used a publish-subscribe mechanism to notify goal violations from multiple sensors to multiple clients. This is similar to mediator pattern listed by Gamma et al. [56]. The decision-making patterns were *adaptation detector*, *case-based reasoning*, *divide and conquer*, *architecture-based* and *tradeoff-based* [36]. Adaptation detector used thresholds for comparison with quantified goal values to detect goal violations and to trigger actions. Case-based reasoning used strategies like Rainbow [9]. Divide and conquer checked a reconfiguration plan, then divided it into multiple actions and ordered these for safe execution. Architecture-based pattern used an architectural model and used the model for reasoning and update . Rainbow [9], MADAM [33] and all architecture-based models seem to use this pattern. Tradeoff-based pattern used optimization methods to choose the best action suited to the context. FUSION [12, 16], MADAM [33] and some other models in the literature used an utility-based model which correspond to this pattern. Four reconfiguration patterns were introduced namely *component inserting*, *component removal*, *server reconfiguration* and *decentralized reconfiguration* [36]. First two patterns dealt with adding and removing components at runtime. Server reconfiguration pattern was used to configure parameters of a server at runtime by storing requests in a buffer. Decentralized reconfiguration pattern was proposed for a distributed environment and imposed responsibility to each component for their insertion, removal and maintaining correct state.

Ramirez et al. applied the design patterns in Rainbow [9]. Although no quantified analysis was provided, it seemed that Rainbow could be modeled with design patterns. They also argued that reusability upto component level was achieved.

The proposed design pattern helps to reuse several components from the man-

aging system and achieves component level reuse. However, as mentioned by Berkane et al, these design patterns were too abstract. It would be useful if these can be matched with GOF design patterns [56] with which the designers are more familiar.

3.6.2 Variability Modeling and Design Patterns for Self-adaptive Systems

Berkane et al. proposed an approach to address variability and reuse by combining product line and design patterns [37]. The design patterns proposed by Ramirez et al. [36] were too abstract to be implemented in a self-adaptive software system [37]. On the other side, product line based methodologies considered variability and commonality for systematic reuse but did not consider recurring problem-based reuse like design patterns did. Reusability can further be improved if two of these can be combined.

Combining design pattern and variability modeling with product lines can lead to some major challenges. Firstly, during domain engineering phase of product lines, design patterns also need to be considered for commonality and variability analysis. This is difficult because design patterns and product lines view reuse from two perspectives. Secondly, as mentioned in the previous paragraph, the higher level design patterns need to be broken down into lower level GOF patterns [56]. However, multiple GOF patterns can be used to specify a higher level pattern. This is why finding the most appropriate one is a challenging task.

Berkane et al. discussed their approach in two stages following product line development which are *domain engineering* and *application engineering* [37]. In the domain engineering phase, an architecture was developed where MAPE-K abstraction [21] was broken down to a list of *logical design patterns* which needed to be followed to develop its monitor, analyze, plan and effect function [37]. These logical design patterns were further broken down to a list of *technical patterns*

which correspond to GOF patterns. In the application engineering phase, the logical patterns to be used were specified from the list according to the application. Then, the technical patterns were selected based on the application domain and selected logical patterns.

Berkane et al. applied their proposed approach on a smart home application. The smart home used sensors to look for events and then triggered an alarm. They designed the domain engineering and application engineering phase with this system, followed by a demonstrative implementation of the design patterns. They measure coupling and cohesion of components to evaluate their approach [37]. However, no comparative study was provided.

The technique proposed by Berkane et al. seem to improve reusability by following design patterns from the product line. However, this approach used only an abstraction like MAPE-K framework for self-adaptation. Thus, it does not seem to address effectiveness of adaptation. Nevertheless, this is understandable that if the design patterns can be combined with an effective self-adaptive system algorithm, it may help to achieve this goal.

3.6.3 Summary of Design Pattern-Based Approaches

As mentioned previously, use of design patterns to address reusability of self-adaptive systems have just begun. The facility of design patterns is that it can help to achieve reuse upto components and subcomponents level which the product line-based approaches lack. However, it is still not addressed in the literature how to use these design patterns in an effective self-adaptive design methodology such as FUSION [16] or a control based system [10, 32]. So, addressing this may result in a reusable and effective self-adaptive system, which is much desired.

3.7 Summary

From the discussion of the existing works in self-adaptive system, it is evident that much work have been done on designing power adaptation component. However, it is also visible that developing reusable self-adaptive systems are still challenging. Besides, hiding the complexities of adaptation logic from the developers of self-adaptive system also seem to be challenging. However, following software engineering principles, the recent approaches to design self-adaptive systems hold promises to pave the way towards mitigating these issues.

As discussed in 3.3.4, self-adaptive systems also lack tool support. Although some tools such as *FUSION* [72], *DiVA Integrated Studio* [73] etc. exist, but these either do not support reuse or lack extensibility. Tools for self-adaptive system development, specifically an adaptation code generation tool can help to achieve systematic reuse. So, researches that focus on developing reusable adaptation components that can be generated with such tools can solve the aforementioned problems.

Chapter 4

A Reusable Adaptation Component Design Technique for Self-Adaptive System

4.1 Introduction

The existing works on self-adaptive software design shows that achieving reuse while ensuring effectiveness is difficult. Architecture-based approaches [9, 60] seem to use static strategies which hinder reuse [33]. Component model-based methods [13, 15] help to achieve reuse but full code refactoring is needed. Control theory-based techniques [59, 32, 10] face problems when the model of the system needs to be updated during reconfiguration. Machine learning-based approaches [12, 16, 11] seem to achieve effective adaptation but reuse becomes difficult. However, recent approaches to self-adaptive software design, namely product line-based and design pattern-based approaches prioritize reuse. However, these does not integrate their model with an effective self-adaptive system design. In this report, a methodology for solving these problems is proposed. The aim of this method is to achieve effective adaptation and systematic reuse.

4.2 Reusable Adaptation Component for Self-Adaptive Systems

To discuss the proposed methodology, it is divided into two parts namely logical view and structural view. In the logical view, the core algorithm and step by step process of generating an adaptation component for a specific system have been discussed. In the structural view, design patterns that enable systematic reuse have been discussed.

4.2.1 Logical View of The Model

In this section, the logical structure of the model along with the algorithms for its components is discussed. To ensure the effectiveness of the adaptation component, an approach similar to FUSION framework [12, 16] has been taken to develop the adaptation logic. This is because, as seen from the literature, FUSION solves the problems of static strategies which can help to increase reuse. Besides, FUSION is efficient because it reduces feature-space.

According to the literature, FUSION did not address reusability and also, the subcomoponents of the adaptation component were not reusable. This is because it did not follow any design patterns to enable separation of concerns between subcomponents. However, the proposed technique addresses these problems. Besides, the proposed methodology also introduces an dynamic AOP-based [15, 57] integration mechanism to automate the interaction between adaptation and business logic. As literature show, this was mentioned to be a research challenge that still remains unsolved [34]. The proposed technique also incorporates additional training features along with adaptation features to provide a better prediction model than FUSION. In the following subsections, the overview of the approach and the specific parts of it have been discussed.

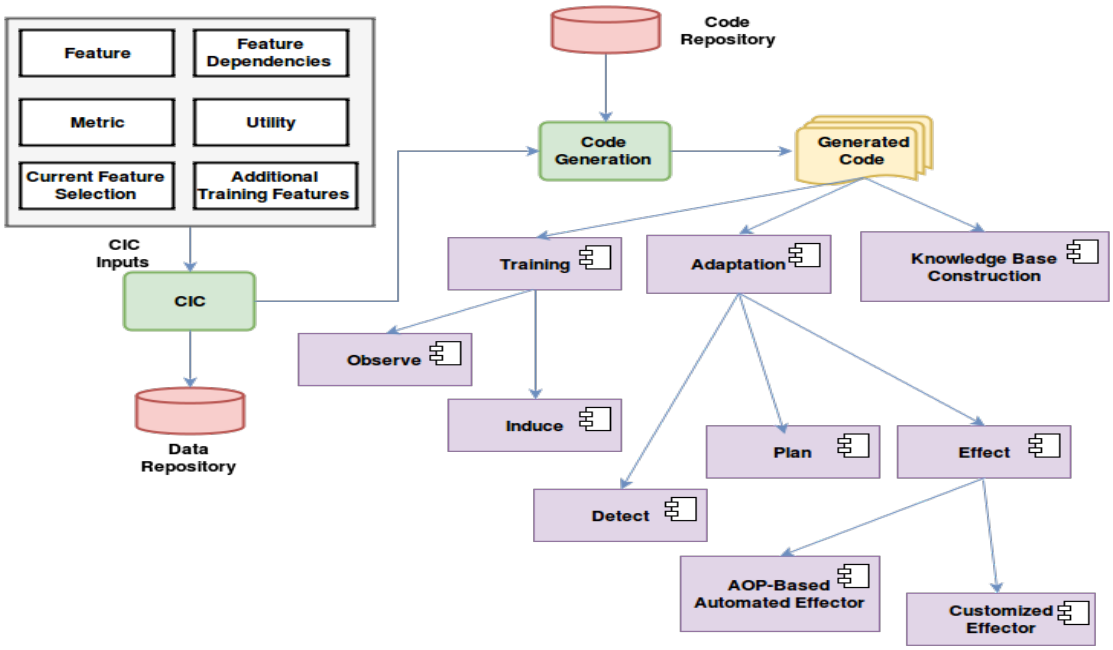


Figure 4.1: The Logical View of the Proposed Methodology

Overview

Figure (4.1) depicts the overview of the proposed method. It consists of two process which are *Configuration Information Collection (CIC)* and *Code Generation (CG)*. CIC is used to collect information about *features*, *feature dependencies*, *current feature selection*, *metric* and *utility functions*, and *additional training features* (features that will be used for training but cannot be turned on or off) from the developers. This information is stored in a repository and used in the CG step. An adaptation logic code is developed beforehand and stored which contains some concrete implementation that can be readily used, and some abstract parts which need to be specialized. In the CG step, the adaptation logic codes are generated by incorporating the information from the CIC step with the aforementioned adaptation logic code base.

The adaptation logic consists of three parts which are *Knowledge Base Construction*, *Learning* and *Adaptation* [12, 16]. The Knowledge Base Construction part constructs the knowledge base by randomly selecting features, calculating metric values for that selection and then, storing these. Learning consists of *Ob-*

serve and *Induce* [12, 16] where Observe normalizes the raw metric values from the knowledge base and induce applies learning to find feature-metric relationships from it in the form of equations. These relationships are used by the Adaptation part that consists of *Detect*, *Plan* and *Effect* [12]. Detect monitors the system continuously by reading utility values to track goal violations. A utility value less than one indicates violation of a goal in this method. In case of a goal violation, Detect invokes the Plan part that uses the equations from the Learning step to identify features that are related to this violated goal or metric. Then, all the metrics that are affected by these shared features are detected from the equations. These detected metrics correspond to conflicting goals. In this case, an optimization problem is generated that considers maximizing total utility of the conflicting goals, subject to feature dependencies. The solution to this optimization problem is a selection of features that maximizes the aforementioned total utility. In Effect, the features are turned on or off, or two features are swapped. The feature selection along with its metric values are stored in the knowledge base for addressing the new pattern. In this way, the knowledge base is gradually enriched. It is mentionable that Effect may add, remove or swap (that is, turn on, off or swap) features with two types of components which are *AOP-Based Automated Effector* or *Customized Effector*. The first one uses dynamic AOP [34] to add, remove or swap components at runtime. The second one uses abstractions developed in the adaptation logic and specify these to add, remove or swap components. The reason of using two approaches, instead of one, is that the first one is applicable when the business codes use factory methods and interfaces for feature initiation and interaction respectively. If this is not the case, second one can be used. As the Learning and Adaptation part interacts with the features of the system through Effect, the whole adaptation logic and the business logic are kept separate and reusability of the generated codes is ensured. In the following sections, the whole process and the components are discussed in details.

Table 4.1: Constraints for Feature Relationships

Feature Type	**Feature Constraint**	**Feature Relation**
Optional	$\sum_{\forall f_n \in \text{zero-or-one-of-group}} f_n \leq 1$	zero-or-one-of-group
Mandatory	$\sum_{\forall f_n \in \text{exactly-one-of-group}} f_n = 1$	exactly-one-of-group
Mandatory	$\sum_{\forall f_n \in \text{at-least-one-of-group}} f_n \geq 1$	at-least-one-of-group
Optional	$\sum_{\forall f_n \in \text{zero-or-all-of-group}} f_n \bmod n = 0$	zero-or-all-of-group
Depends on child features	$\forall \text{child} \in \text{Shared Features} f_{parent} - f_{child} \geq 0$	parent child relation

Configuration Information Collection

In this step information about the system is collected. All the information that are collected is shown in Figure 4.2. These information are available because the business logic is assumed to be already developed. Information about The features and the classes that implement those are collected. Feature dependency information which are dependent features and their dependency types are collected. Esfahani et al. mentioned five dependency types which are mentioned in table 4.1. [12]. These dependency types are described below.

1. *zero-or-one-of-group*: This resembles that more than one features cannot be enabled.

2. *exactly-one-of-group*: This means that exactly one feature can be enabled at a time in the feature group.

3. *at-least-one-of-group*: This dependency indicates a mandatory relationship where at least one of the features in the group must be enabled.

4. *zero-or-all-of-group*: It indicates that either all or none of the features will be turned on.

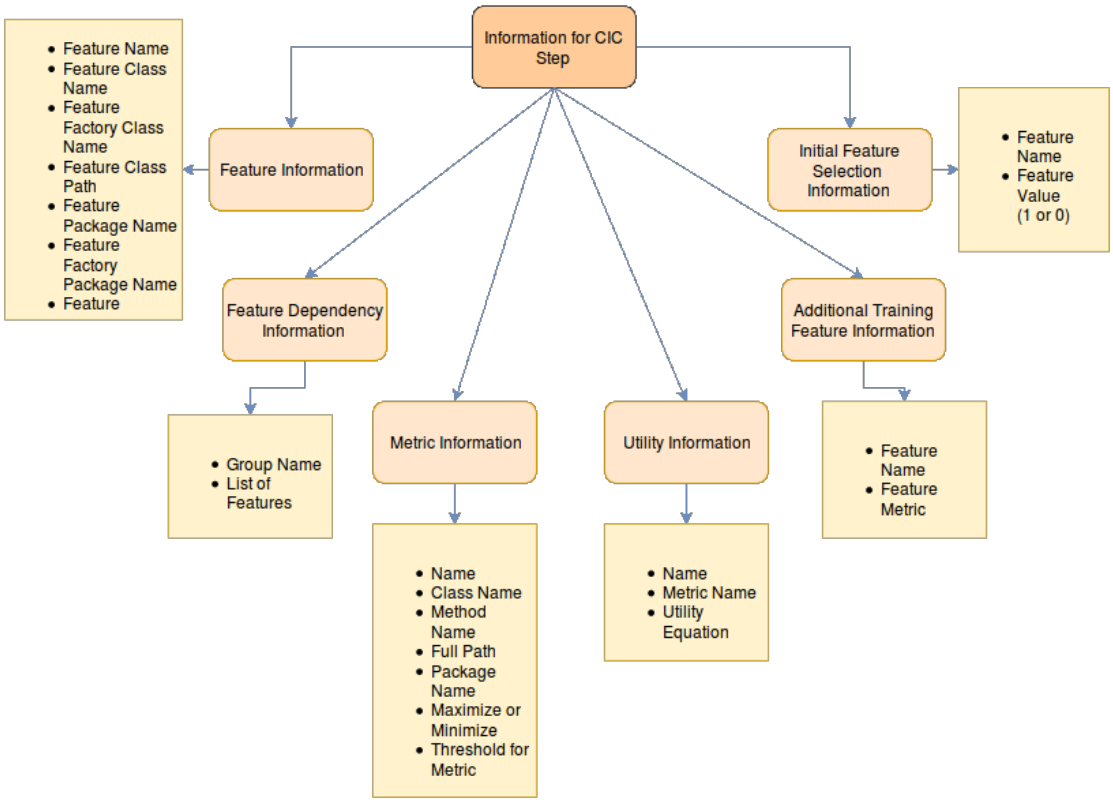


Figure 4.2: Configuration Information

5. *parent child relation*: This means that enabling a specific feature (parent feature) requires all other features of the group to be enabled.

This step also collects metric and utility information. In case of metric information, developers need to specify metric names, metric types (that is, whether the metric needs to be maximized or minimized for goal conformance) and the classes and methods that calculate these. The proposed methodology uses the metric types (maximize or minimize) to generate the utility equation. For maximization goals, where the metric values need to be more than the provided threshold, the generated equation has the form similar to Equation 4.1.

$$Utility_n = Metric_n - Threshold_n \tag{4.1}$$

Where $Utility_n$ is the utility value, $Metric_n$ is the metric value and $Threshold_n$ represents the metric threshold value for the nth metric.

For minimization goals, the metric values should never cross the provided threshold value. For this, if a minimization type is provided, an equation such as Equation 4.2 is produced.

$$Utility_n = Threshold_n - Metric_n \tag{4.2}$$

These equations are stored as utility equations with the utility information. It is notable that, the existing system needs to expose an API for metric calculation which is a common assumption in self-adaptive system [12, 9]. This is also a valid assumption because there is no known way to automatically connect the system to an appropriate metric calculation logic without analyzing its context manually. For utility information, the utility names and corresponding metric names are required.

The current feature selection is provided as a name-value pair where the value is either 1 or 0 (enabled or disabled respectively). The current feature selection is needed when the system starts for the first time and an initial feature selection is required for knowledge base construction. The additional training features, which are features that are needed to accurately predict a specific goal, are also given. It is notable that these features must have a corresponding metric provided in the aforementioned metric information collection step. The metrics calculate current values for these additional features which are used for both knowledge base construction and adaptation. As an example, consider a scenario where performance needs to be adapted. To predict the performance goal value, service time, requests per second etc. can be used. These predictors or features cannot be turned on or off from the application for being properties of the server machine. However, these must be incorporated for better prediction of goal values. While generating the previously mentioned optimization problem, the current values of these must be considered for better accuracy of the solution. This is why these additional

training features are mandatory for a more effective adaptation mechanism. It is notable that the consideration of additional training features is absent from the FUSION framework.

It is to be mentioned that all the properties mentioned in Figure 4.2 is not always required. For features, in case of AOP-Based Automated Effector, all the properties are required. For other customized effectors, only the feature name is needed and the other fields can be left blank. In the metric information, the class and package information are required only when a class based metric API has been provided by the business logic, that is, the metric information can be obtained by method calls from classes. In other cases, where metric API is not a class, only the metric name and full API path need to be provided. All these information are stored in a data repository for further use in the subsequent steps.

Code Generation

This step produces the adaptation logic codes and provides support for integrating adaptation and business logic components. Adaptation component along with its specific parts and abstract hooks are developed and stored in the code repository. Code generation is a task of replacing and updating configuration elements of these codes using the information from the data repository. The following subsections discuss the adaptation logic in this CG step in details. The adaptation logic codes, as mentioned before, consist of three types of components which are Knowledge Base Construction, Learning and Adaptation. Here, the Learning and the Adaptation step follow a similar approach discussed by Esfahani et al. [12]. However, additional training features have been incorporated with the FUSION model to provide more effective prediction.

Knowledge Base Construction When the application starts for the first time, the knowledge base is empty. In this case, a Knowledge Base Construction compo-

nent is provided. This component can be run individually. To use this, the system needs to be put in an operational environment, real or simulated. After running this component, it will randomly select features and these selections are stored in the knowledge base along with the corresponding metric values. Thus, a primary knowledge base is constructed which is later used for Learning and Adaptation.

It is mentionable that machine learning methods do not work if there is no data. The Knowledge Base Construction helps to solve this problem. The only assumption is that, the system needs to be put in an environment that represents its real execution environment. Often, this is not possible but creating a simulation of the operation environment is possible with modern tools in hand. For example, Apache JMeter [74] is a tool that can simulate system load with multiple users. This is why the assumption is valid.

To generate data for knowledge base, at first the feature groups are considered. For each feature group, features are selected randomly (turned on or off randomly) while maintaining the constraints mentioned in Table 4.1. These feature groups are merged to get a full feature selection set. Then, for each of the metrics, the metric values are collected. It is mentionable that the additional training features, as mentioned previously are also calculated by their corresponding metric values. The feature selection and metric values are then joined and thus, a data is generated. This process is continuously run in a simulated environment to gradually enrich the knowledge base.

Learning Learning component consist of Observe and Induce components. These two are described below.

1. Observe: It normalizes or standardizes the raw metric values from the knowledge base. Normalization brings all metric values within the same range (generally 0 and 1). For *normalization* or *standardization*, either of Equa-

tion 4.3 or 4.4 can be used [16, 12, 52].

$$x_{norm} = \frac{x - x_{minimum}}{x_{maximum} - x_{minimum}} \tag{4.3}$$

$$x_{norm} = \frac{x - \mu}{\sigma} \tag{4.4}$$

Here, in Equation 4.3, x_{norm} represents normalized value of x and $x_{maximum}$ and $x_{minimum}$ represent minimum and maximum values of x respectively. In Equation 4.4, μ is the average and σ is the standard deviation of the data. Observe also checks if the knowledge base contains all the patterns of the environment as training data that are required for accurate adaptation. This is checked by observing the number of failed adaptations and checking whether this exceeds a predefined threshold In this case, a new pattern is considered to have appeared and learning is conducted again.

2. Induce: Induce does two tasks which are *feature selection* and executing the *learning algorithm*. First, a *significance test* [12, 16] over the knowledge base is performed. Significance test selects those features for each of the metrics that have a significant influence on these. Using these features, a learning algorithm derives the feature-metric relationship equations. Linear regression algorithm [52] has been used as the learning algorithm because it converges fast [16]. A typical output from this training phase is like Equation (4.5).

$$Metric = k_1 \times Feature_1 + k_2 \times Feature_2 - k_3 \times Feature_3 - k_4 \times AdditionalFeature_1 \tag{4.5}$$

Where k_1, k_2 etc. indicate constants, $Metric$ is any metric and $Feature_1$, $Feature_2$ etc. are feature state which can be zero or one (enabled or disabled respectively). $AdditionalFeature_1$ is the additional feature considered for training and which cannot be turned on or off. It is also mentionable that for

using the calculated value from this equation, it needs to be denormalized using a reverse process of normalization [12, 16]. This is because without *denormalization*, it will not represent values in the real range of metrics.

Adaptation

The adaptation part is comprised of three components which are *Detect*, *Plan* and *Effect*. These components are mentioned below.

1. Detect: The system needs to adapt whenever there is a violated goal. Violated goal is detected with the help of utility functions. From Equation 4.1 and 4.2,it is clear that the utility function return a values less than one in case of a goal violation. This property is used to detect goal violations in the proposed technique. These violated goals are passed to the next stage namely Plan.

2. Plan: Detection of goal violation invokes Plan component. At Learning, feature-metric relationships have been identified. These relationships are now used to find the related features with the violated goals which are called *shared features* [12, 16]. These features may also affect other goals which are known as *conflicting goals* [12, 16]. These are conflicting because if a selection of features affects one of these goals, the other ones are affected too. The system needs to select a set of features that maximize its utility given the corresponding conflicting goals and shared features. Equation (4.6) can be used for this purpose.

$$F = maximize \sum_{\substack{F_s \in \text{shared features} \\ \forall \text{conflicting goals t}}} U_t(M_t(F_s)) \tag{4.6}$$

However, as additional training features are used to generate feature-metric equations in the Learning component, The variables for the additional train-

ing features also occur in the utility function equations. As these training features are external to the system and cannot be selected or turned on or off, the current training feature values are collected at runtime using their corresponding metrics. This current values are used in place of their representation variables in all the equations.

From the above discussions, it is evident that an optimization problem needs to be constructed for planning where the features can be represented by 1 or 0 (on or off). So, this can be solved using *Integer Linear Programming* [69] with binary variables. However, none of the utility functions should have value less than one, because it will trigger goal violations. The system also must not violate any feature dependencies mentioned previously while optimizing. Besides, as mentioned in the previous paragraph, the additional training feature values need to be equal to their current values. So, if f_1 and f_2 are shared features and f_{ex} is the additional training feature, a valid optimization problem would be as mentioned in Equation (4.7).

$$
\begin{aligned}
& F_{selection} = maximize(U_{t1}(M_{t1}(F) + U_{t2}(M_{t2}(F)) \\
& \textbf{Subject To} \\
& U_{t1}(M_{t1}(F) > 0 \\
& U_{t2}(M_{t2}(F) > 0 \\
& f_1 + f_2 = 1 \\
& f_{ex} = 2 \\
& \textbf{Where} \\
& M_{t1}(F) = 2.77 \times f_1 - 1.75 \times f_2 + f_{ex} \\
& M_{t2}(F) = 2.11 \times f_1 + 3.35 \times f_2
\end{aligned}
\tag{4.7}
$$

$F_{selection}$ is the selected feature string after the optimization problem is solved. This feature selection string indicates the features needed to be turned on or off or swapped to satisfy all goals as much as possible.

3. Effect: This component executes the feature selection into the system. Effect can be done with two components as mentioned previously which are AOP-Based Automated Effector and Customized Effector. In AOP-Based Automated Effector, at first, the interface information of every feature is gathered. To turn off a behavior, a *mock object*, which is an object that simulates the behavior of a real one, is created and returned from the factory method of that feature. The mock object only simulates the behavior but do not perform any actions and so, the feature is effectively turned off. To turn on a feature, reflection [13] is used to instantiate the feature class at runtime. Then, dynamic AOP is used to intercept the return value of a feature factory method to return this instantiation. To swap two features, the instance of the one that will be turned on is returned from both of the factory methods with the help of the around advice [57]. However, this technique assumes that the instantiation of features have been kept separate from the business logic using factory method pattern. The dependent features also need to be coded to the interface which is a very common practice.

   From Table 4.1, features can be optional or mandatory. Although in ideal case, optional features should be toggled without any side effect, often optional features show dependency with other operational components due to implementation issues. This is known as Optional Feature Problem [75]. To address this issue, the Customized Effector is introduced. Customized effectors provide hooks like product line-based approaches to specialize these to the application. Generally, the developers need to add codes for adding, removing and swapping components by extending an abstraction.

As features are interdependent, while enabling or disabling these, the dependency constraints must be maintained. For this purpose, after the feature selection string is produced, for every feature, it is scanned to find any violation of constraints mentioned in Table 4.1. Any violation represents a new pattern. This feature selection string is rejected and learning is again performed to capture this pattern.

It is evident from the above discussion that a reusable adaptation component can be generated following the proposed methodology. Although some specialization may be needed, it is much smaller than refactoring the full codes. So, along with effective adaptation following FUSION [12, 16], reusability is also ensured. In the next section, the structural view of this model is discussed which ensures subcomponent level reuse.

4.2.2 Structural View of The Model

In the structural view, the subcomponents of both learning and adaptation components have been organized with GOF design patterns [56], so that these can be reused or customized easily. In the Learning component, from Figure 4.3, Decorator pattern has been used for prepossessing before applying the learning algorithm. Normalization and feature selection have been used as decorators to preprocess the training data in the knowledge base. Different learning algorithms have been represented by the Strategy pattern. Decorator pattern has been used between Preprocessing and Learning because each process transforms data without completely relying on the other which clearly indicates that decorator pattern should be used. Finally, the Observer pattern has been used between *Threshold-BasedObservee* and the Decorator interface for observing training inaccuracy and re-perform training.

In the adaptation component, from Figure 4.4 Observer pattern is used between detect and plan interface (Monitoring and Analyzer). Decorator patter

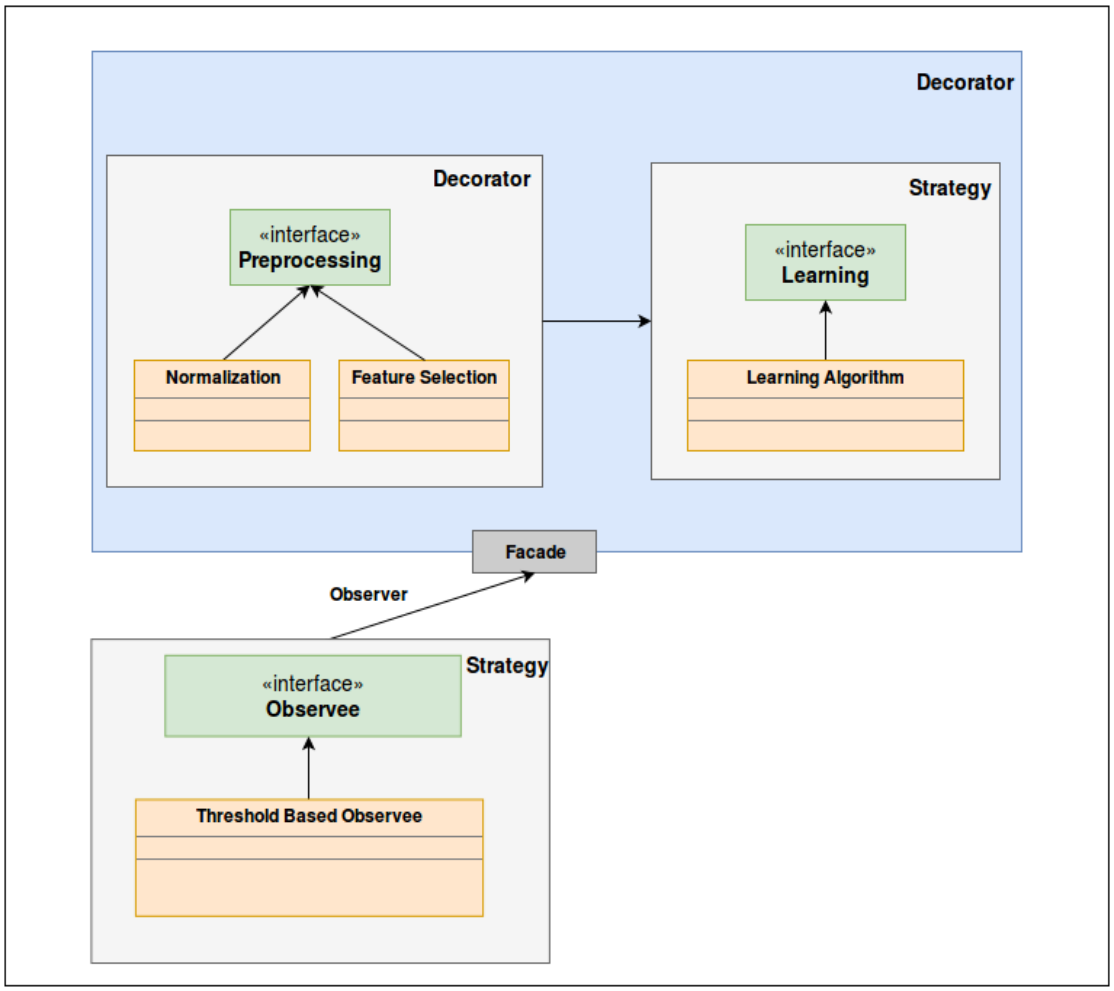


Figure 4.3: The Structural View of Learning Component

have been used to gradually decorate the optimization problem with constraints and the objective function. To support multiple optimization algorithm, optimization solver was designed with Strategy pattern. Between Effect and Plan interface, Observer pattern has been used. Finally, effectors have been integrated with feature selection validator using command pattern. In the following subsections, all these design patterns and their applications in the proposed approach have been discussed in details.

Learning

Learning component contains strategy, decorator and observer patterns. The structural view of these have been shown in Figure 4.3

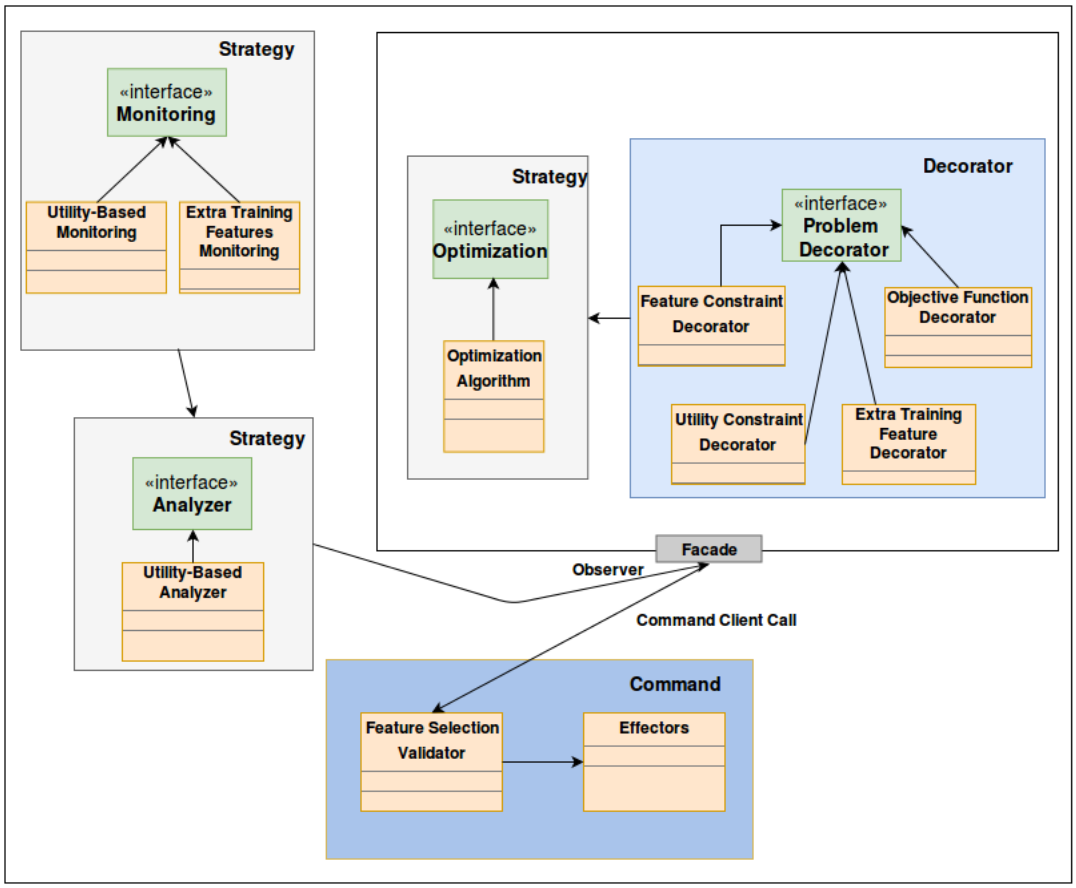


Figure 4.4: The Structural View of Adaptation Component

1. Strategy Pattern: Strategy pattern is used when algorithms need to be varied independently. Adding an algorithm involves only implementing the abstraction and passing its reference to the invoker of the algorithm. Thus, it promotes customizability and reuse. This is why this pattern has been used to support reuse and customization of the learning algorithm and observe mechanism.

2. Decorator Pattern: Decorator patter is used when functionalities need to be added dynamically. Decorator pattern has been used for the preprocessing logic, that is, to pass the normalized data to the feature selection component and then pass the selected features to the learning algorithm. Decorator pattern helps to add any object within the flow easily. For example, if anyone wants add missing data handling, he needs to create a class and insert

it between normalization and feature selection which involves changing only two references.

3. Observer Pattern: This is used when a notification scheme is needed. For this, it has been used to notify the learning process to start again because a new pattern has arrived. To achieve this, an interface need to be present between normalization, feature selection, learning algorithm combined, and the observee.

4. Facade Pattern: This pattern provides a single interface to a set of classes or interfaces. This pattern has been used to construct an interface with normalization algorithm, feature selection and learning because it it necessary for communicating with the threshold-based observee through observer pattern.

Adaptation

This component uses Strategy, Observer, Decorator, Command and Facade patterns. The uses of these patterns in the adaptation component is described below.

1. Strategy Pattern: This is used to support customizability and reuse of different optimization algorithms, as mentioned previously.

2. Observer Pattern: This is utilized to route goal violation indication to the Plan interface which is a facade interface.

3. Decorator Pattern: The Decorator pattern has been used to add attributes to the optimization problem when adaptation is required. The feature constraints, utility constraints, additional training feature constraints and the objective function is added to build a complete optimization problem. Then, it is passed to an optimization algorithm for receiving an optimal feature selection. As the problem is build at runtime gradually, Decorator pattern is

suitable for this purpose.

4. Command Pattern: Command pattern is used when request need to be queued and for separating caller and receiver of a request. So, this is used to capture the feature selection from Plan facade interface, analyze it and pass to the effectors only if it is valid. This helps keeping the effectors and Plan components separate.

5. Facade Pattern: It has been used to produce an interface between Detect and Plan, and an interface between Plan and Effect.

It is mentionable that the design patterns which are mentioned in the previous sections were chosen based on analyzing the existing works on self-adaptive system design. Applying design patterns for self-adaptive system design helps to maintain a consistent structure of the adaptation component. For this reason, adding or removing any part of the adaptation component becomes easier. The proposed methodology incorporates design patterns as a part of the adaptation component design mechanism to ensure that systematic reuse can be achieved.

4.3 Summary

From the above discussion, it is evident that the proposed approach addresses both reusable and effective adaptation. The machine learning based technique discussed in 4.2.1 can help to achieve adaptation which is efficient and effective. This technique helps to generate codes for any application if the mentioned inputs are provided. However, it does not guarantee that different subcomponents of the adaptation logic are reusable. This is achieved through design patterns, as discussed in 4.2.2. Thus, the proposed technique improves on the existing techniques in the literature by ensuring both component and subcomponent level reuse while preserving the effectiveness of adaptation.

Chapter 5

Implementation and Result Analysis

A reusable self-adaptive system design has been a research challenge as seen from the literature [12, 5, 4]. A reusable adaptation component design mechanism was proposed in the previous chapter. The goal was to improve the overall reusability of the adaptation component while preserving the effectiveness of adaptation. In this chapter, the result analysis for the proposed mechanism has been shown. The proposed method was validated by performing a case study on Znn.com [17, 4]. The model was tested using reusability metrics such as *Lines of Code (LOC)*, *Message Passing Coupling (MPC)* [2] and *Lack of Cohesion of Methods 4 (LCOM4)* [3]. The code generated by the mechanism was deployed in an operational environment for Znn.com. In this environment, five servers were connected to a load balancer and the main goal was to keep the response time within a specific limit while preserving a minimum content quality and a maximum cost. The response time was observed in a high load situation to see if adaptation occurs. In all the cases, the proposed method ensured reusability and effectiveness for the performed experiments .

5.1 Implementation Details

The proposed methodology was implemented in Java using Eclipse Mars 2 Integrated Development Environment (IDE) [76]. An Object Oriented Programming language such as Java provides extensive support for polymorphism, inheritance and encapsulation which are necessary for reusability. This is why it was used to implements the approach. The following tools and libraries were used to develop the system.

- Weka 3.7.3 [77]: Weka is a data analysis tool that provides support for preprocessing and analyzing data. Weka also provides API to access different training algorithms such as linear regression, M5P etc. Weka was used to provide a simple structure to the Training component of the algorithm.

- EvalEx [78]: This is a simple equation evaluator for Java. This was used to evaluate the value of the utility from the utility equation.

- Commons IO [79]: Commons IO or Apache Commons IO is a library that implements Input-Output (IO) functionality for Java based systems. This library was used to access and modify files.

- JUnit 4.12.0 [80]: JUnit is a library that helps to write unit tests in java. So, it was used to write unit tests for the modules in the system.

- Mockito 1.9.8 [81]: Mockito helps to create mock objects which is useful to simulate behavior in JUnit tests. This library was also used for this purpose in the unit tests of the system.

The implementation was executed on an operating system with following configurations.

- Operating System: Ubuntu 14.04 LTS

- RAM: 8.00 GB

Listing 5.1: Feature XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <feature>
        <featureName>feature1</featureName>
        <featureClassName>FeatureClass1 </featureClassName>
        <featureFactoryClassName></featureFactoryClassName>
        <featureFactoryMethodName></featureFactoryMethodName>
        <featureClassPath>/home/path/FeatureClass1</featureClassPath
            >
        <featurePackageName>org.path.project</featurePackageName>
        <featureFactoryPackage></featureFactoryPackage>
        <featureFactoryClassPath></featureFactoryClassPath>
    </feature>
    <feature>
        <featureName>feature2</featureName>
        <featureClassName>FeatureClass2</featureClassName>
        <featureFactoryClassName>/home/path/FeatureClass2</
            featureFactoryClassName>
        <featureFactoryMethodName></featureFactoryMethodName>
        <featureClassPath></featureClassPath>
        <featurePackageName>org.path.project</featurePackageName>
        <featureFactoryPackage></featureFactoryPackage>
        <featureFactoryClassPath></featureFactoryClassPath>
    </feature>
</configuration>
```

- CPU: 3.30GHz Intel Core i3 Processor
- Platform: 32 bit

To execute the implementation, attributes of features, metrics, utilities, feature dependencies, training features and current feature selection need to be provided. These information are collected using the tool interface of the implementation. The tool writes these in the xml files as shown in Listing 5.1-5.6 . It is seen from the feature XML file in 5.1 that feature name, class name, factory class name, factory method name, class path, package name, factory package name and feature factory class path can be provided. However, all these information is needed only if AOP based effector is used. If customized effectors need to be used, only feature name needs to be mentioned and other fields can be left blank.

Listing 5.2: Metric XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <metric>
        <name>metric1</name>
        <className>MetricClass1</className>
        <methodName></methodName>
        <fullPath>/home/path/MetricClass1</fullPath>
        <packageName>org.project.metric</packageName>
    </metric>
    <metric>
        <name>metric2</name>
        <className>MetricClass2</className>
        <methodName></methodName>
        <fullPath>/home/path/MetricClass2</fullPath>
        <packageName>org.project.metric</packageName>
    </metric>
    <metric>
        <name>metric3</name>
        <className>MetricClass3</className>
        <methodName></methodName>
        <fullPath>/home/path/MetricClass3</fullPath>
        <packageName>org.project.metric</packageName>
    </metric>
</configuration>
```

Listing 5.3: Feature Dependencies XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <featuredependency>
        <group>exactly one of group</group>
        <feature1>feature1</feature1>
        <feature2>feature2</feature2>
    </featuredependency>
</configuration>
```

From the metric XML file in 5.2, it is seen that metric name, class name, method name, full path and package name can be given. In the implementation, two types of metric calculation scheme is supported which are class-based and url-based. The XML file shown in Listing 5.2 is for a class-based metric calculation scheme. Class-based metric calculation scheme demands a method from the

Listing 5.4: Utility XML File

```
<?xml version="1.0" encoding="UTF 8"?>
<configuration>
    <utility>
        <name>utility1</name>
        <metricName>metric1</metricName>
        <polynomial>0.5 metric1</polynomial>
    </utility>
    <utility>
        <name>utility2</name>
        <metricName>metric2</metricName>
        <polynomial>3+metric2</polynomial>
    </utility>
</configuration>
```

Listing 5.5: Initial Feature Selection XML File

```
<?xml version="1.0" encoding="UTF 8"?>
<configuration>
    <featureselection>
        <featureName>feature1</featureName>
        <value>0</value>
    </featureselection>
    <featureselection>
        <featureName>feature2</featureName>
        <value>1</value>
    </featureselection>
</configuration>
```

business logic to calculate the metric value. Url-based scheme reads metric values from a specific url. It is clear that, for url-based scheme, class name, method name and package name are not required and so, these can be left blank in the metric information.

For utilities, as shown in Listing 5.4, the utility name, the corresponding metric and threshold value need to be provided. From the utility XML file, it is evident that the threshold is converted to a utility equation to be used in the adaptation algorithm as mentioned in the Methodology chapter. In case of feature dependencies, as given in Listing 5.3, the group name and features under the dependency groups need to be given. Listing 5.6 displays the training features, which are extra features that will be considered for training along with the previously mentioned

Listing 5.6: Training Feature XML File

```
<?xml version="1.0" encoding="UTF 8"?>
<configuration>
    <trainingfeature>
        <featureName>trainingfeature1</featureName>
        <featureMetric>metric3</featureMetric>
    </trainingfeature>
</configuration>
```

features. The extra feature name and its corresponding metric are provided in the XML file. Finally, for the current feature selection in Listing 5.5, the feature name and its selection status, that is whether it is on or off, is mentioned by writing 1 or 0 respectively with the corresponding feature.

5.2 Case Study: Znn.com

Znn.com is a model problem which was added by Cheng et al. in the exempler repository of the Software Engineering for Self-Adaptive Systems community [17]. The Znn.com system has been used in numerous papers [4, 82, 83, 61, 84] for evaluating the adaptation approaches. This is why Znn.com has been used to assess the proposed methodology. In this section, the architecture and adaptation challenges of Znn.com will be discussed.

Znn.com is a news serving application which provides textual and multimedia based news to its users. According to the Znn.com specification, it follows a N-tier style where a load balancer is connected to a server group. The server group can contain one to many servers. The clients send their request to the load balancer and it distributes the requests among the servers in the server group. The whole architecture is shown in Figure 5.1.

The business goals of Znn.com is related to performance, content fidelity or quality and server cost. The main goal is to provide service with a minimum content fidelity and within the budget while maintaining a minimum performance.

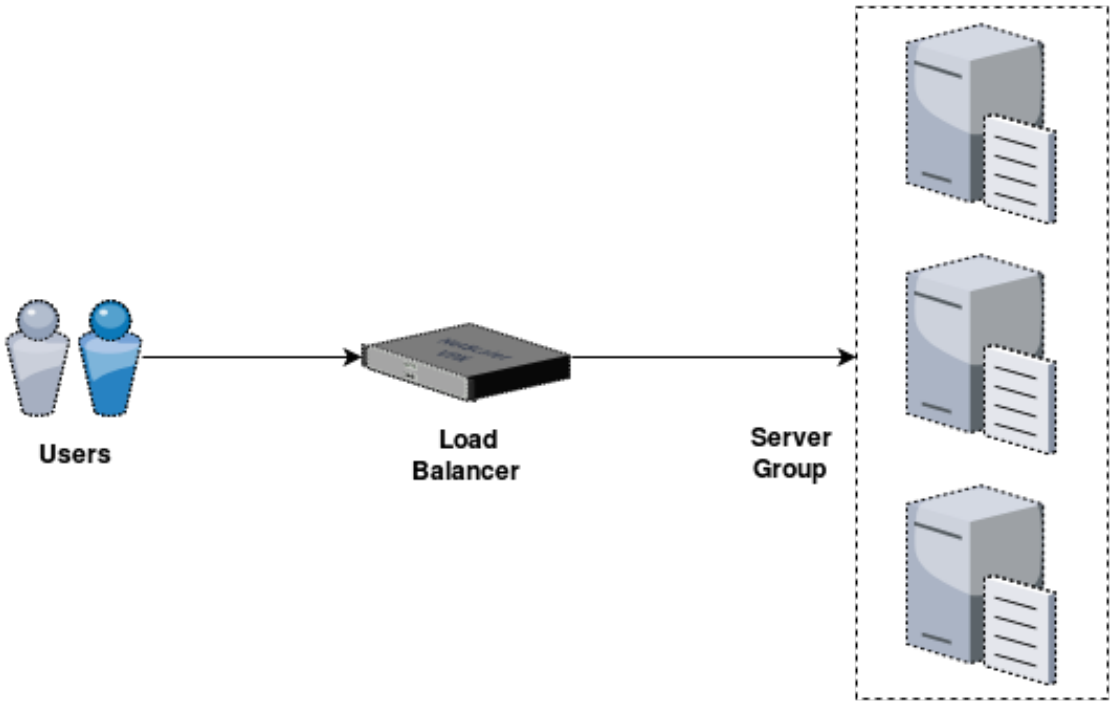


Figure 5.1: N-tier Architecture of Znn.com

However, these goals are related to one another. For example, if content fidelity gets higher, server performance will decrease because the response size is larger for serving higher quality contents. So, a new server needs to be added from the server group. However, servers cannot be added infinitely because the total cost of all the added servers must fall within a specific range. All these scenarios make satisfaction of multiple goals a nontrivial task. This is why Znn.com provides the scope to incorporate a self-adaptive mechanism to optimally work under multiple goals.

Another scenario where Znn.com demands adaptation is when the news website is under a high load situation, also known as Slashdot Effect. As mentioned by Cheng et al. [4], if a website is featured in slashdot.org [85], it gets crowded with visitors within a few hours or days. Due to hit from multiple users, the website might be temporarily down. To partially solve the scenario, some applications such as Gmail request the users to reload later when such a high load situation is detected. However, this is not expected because it hampers the service level of the

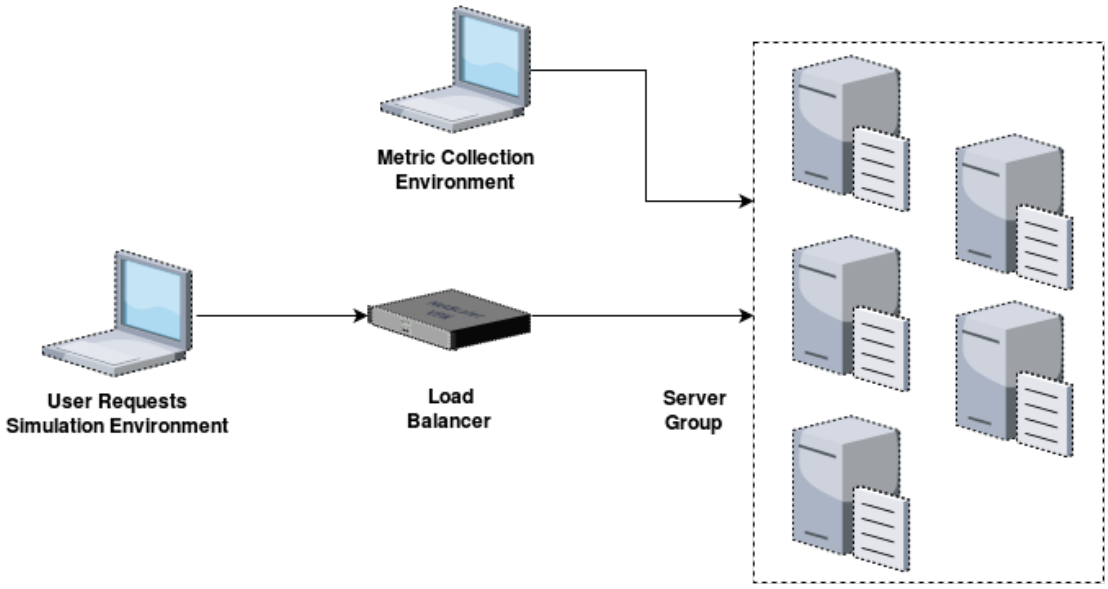


Figure 5.2: Deployment Diagram of Znn.com

application. For this reason, a self-adaptation scheme is required which repairs the system and brings it closer to the three goals mentioned in the previous paragraphs.

5.3 Experimental Setup

Znn.com was deployed on five virtual machines which were connected with another virtual machine acting as a load balancer. Two more virtual machines were used where one helped to collect metric information and the another helped to simulate user requests. The overall architecture of this deployment environment has been shown in Figure 5.2. Each of the virtual machines had the following configuration.

- Operating System: Ubuntu 14.04 LTS
- RAM: 512 MB
- CPU: 3.30GHz Intel Core i3 Processor
- Platform: 32 bit
- Virtual Disk: SATA Controller 8 GB

The implementation of Znn.com was done with PHP and MySQL. The effectors were written in Bash scripting language [86]. In each of the server machines, apache2 web server was used to deploy Znn.com. Apache JMeter was used to simulate user requests in the user requests simulation environment. In the metric collection environment, PHP codes were deployed using apache2 web server which were used to calculate the metric values of performance, cost, content fidelity and some additional training features.

In the simulated environment, response time was calculated using Queueing Theory where the M/M/c queue model was used [87]. A system where multiple requests arrive and are distributed among c servers can be represented by the M/M/c queue model. As calculating real response time from the environment may cause high network overhead and so, may not show the effectiveness of the proposed technique, the response time assumption from the M/M/c queue model was used. The Queueing theory based response time calculation utility provided in the Rainbow framework implementation in [17] was used in this purpose. It receives request arrival rate, service time, content fidelity and number of active servers as input and provides the response time. This indicates that arrival rate and service time can be used as additional training features.

Before starting the experiment, the features and feature dependencies of Znn.com were specified. As features are entities that can vary and can be toggled (turned on or off), it is evident that, every server is a feature. This is because adding a server means turning it on and removing means otherwise. According to Cheng et al., content fidelity has three types which are high, low and text [4]. Each of these are features because these can be toggled. It is also noticeable that at least one of the servers must be turned on to serve contents. This is why the server features belong to *at-least-one-of* dependency group. Besides, exactly one of the content fidelity features can be selected and so, these belong to *exactly-one-of* feature dependency group.

After choosing features and feature dependencies, metrics and utilities were chosen. Response time, content size and number of active servers were used to calculate performance, content fidelity and cost respectively. The threshold for each of these which are maximum response time limit, minimum content fidelity and maximum number of active servers respectively, were chosen. It is understandable that this threshold will vary in different system and in different context because the environment in which a software operates in, is vastly dynamic. For additional training features, service time and request arrival rate were chosen. Finally, current feature selection was provided in the format mentioned in Section 5.1. Using all these information, adaptation logic codes were generated and assessed for reusability and effectiveness of adaptation.

In order to access effectiveness, the system was put under a situation representing the Slashdot effect. To do this, the experiments provided in [88] were repeated. However, each of the experiments was tuned down to around 15 minutes and load five times higher than the mentioned experiment was provided which is mentioned below.

- 15 seconds of load with 30 visits/min
- 2.5 minutes of ramping up to 3000 visits/min
- 4.5 minutes of fixed load to 3000 visits/min
- 9 minutes of ramping down to 60 visits/min

The situation was simulated using the Throughput Shaping Timer plugin of Apache JMeter. The Gaussian Random Timer of JMeter was also used to provide short delay within requests to represent real life behavior.

5.4 Metrics

For assessing reusability of the proposed approach, three metrics were used which are *Lines of Code (LOC)*, *Message Passing Coupling (MPC)* [2] and *Lack of Cohesion of Methods 4 (LCOM4)* [3]. LOC was used in the Rainbow framework by Cheng et al. [88] for assessing reusability. However, LOC, as a measure of reusability has been criticised in some of the literature [89, 4] because LOC does not represent the connections between and within the classes or modules. This is why coupling and cohesion based metrics were used. It has been seen that reusability depends on coupling and cohesion of classes [90] as low coupling and high cohesion increases the chance of reuse [90]. For this reason, MPC and LCOM4 were used to measure the reusability of the approach. The definition of these metrics are given below.

- LOC: LOC counts the number of lines in a code. However, a number of issues such as, whether comments, blank lines etc. will be considered, becomes a concern. David Wheeler developed a code analysis tool named *SLOCCount* [91], which was also used by Rainbow for counting LOC. In this tool, an LOC is considered as a line terminated with a newline and which contains at least one character excluding whitespaces and comments. To validate the proposed methodology, SLOCCount was utilized to calculate LOC. The lower the LOC, the higher the probability of reuse.

- MPC: According to [89], MPC is a valid measure of coupling and so, a valid measure of reusability [89]. MPC indicates the number of external invocation of methods from a class. For example, if a class calls 5 methods of some other classes, the value of MPC for this class is 5. MPC was used to measure the coupling between the classes of the adaptation logic, after it was integrated with Znn.com. The higher the MPC, the higher the class is dependent on other classes and so, the lower the reusability.

- LCOM4: LCOM4 is a measure of cohesion. Cohesion indicates the strength of internal relationships of functionalities within a class. LCOM4 is the number of *'connected components'* within a class [92]. A connected component consists of a group of methods which either call one another or share at least one instance variable of the class. Presence of multiple connected components for a class means the class performs multiple unrelated responsibilities. This is why cohesion and reusability increases as LCOM4 decrease. The ideal values of LCOM4 are either 0 or 1 [92].

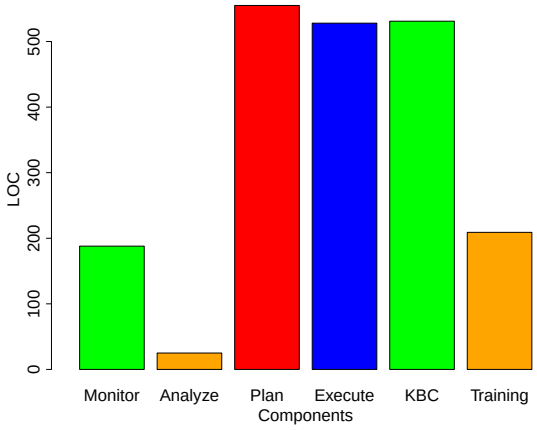
To assess the effectiveness of adaptation of the methodology, the experiment mentioned in Section 5.3 was performed five times starting from a single server and high fidelity feature selection. This is because this feature selection results in the worst performance. Every time one of the five servers is chosen and the load is increased by any constant factor. In the experiments conducted, the load was increased by 120 visits/min and it was seen that he system reaches its maximum capacity after five runs. This is why the experiment was performed five times to test for adaptation quality. In each of the runs, it was observed whether the proposed methodology could gradually improve performance. Same as the adaptation logic design validation experiments from the literature [4, 12, 11], the value of the main objective, response time in this case, was compared in two situations, namely adaptation and without adaptation.

5.5 Result Analysis

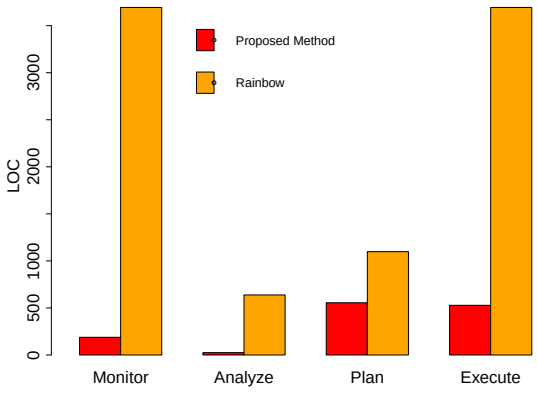
The proposed methodology was applied to generate code for Znn.com, as mentioned previously. The generated code was analyzed based on the metrics mentioned in the previous section. As Rainbow used LOC as a measure of reusability for the same case study on Znn.com, the LOC of the proposed model is compared to that of Rainbow. The MPC and LCOM4 values are compared with ideal values

Table 5.1: Proposed Method vs. Rainbow Considering LOC

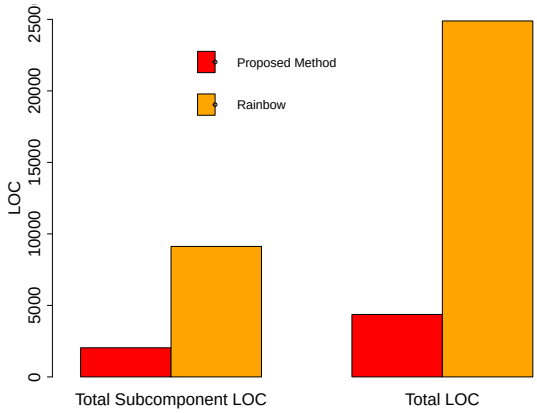
Approach	LOC						Aggregated LOC	
	Monitor	Analyze	Plan	Execute	KBC	Training	Subcomponent	Full Code
Proposed Method	188	25	555	528	531	209	2036	4367
Rainbow	3694	638	1098	3694	-	-	9124	24891



(a) LOC of Sub-Components



(b) Sub-Component LOC for Rainbow vs. Proposed Method



(c) Total Sub-Component and Full Code LOC

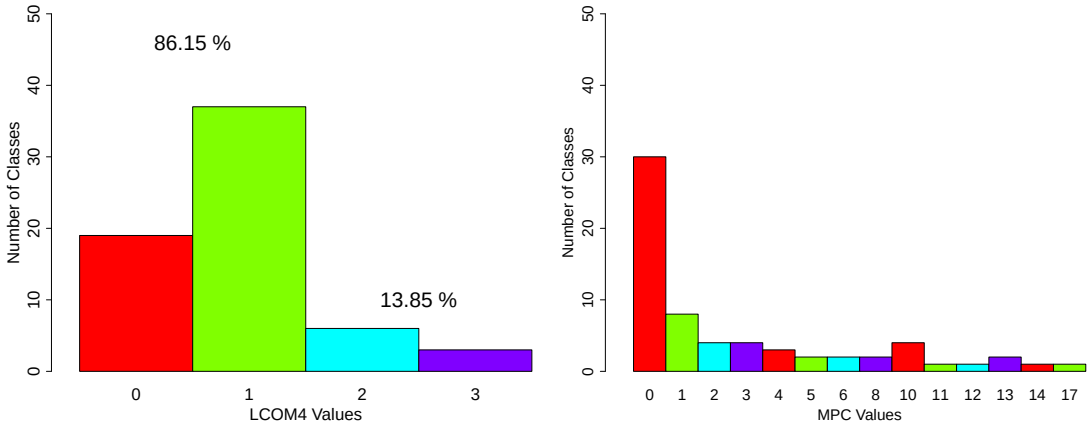
Figure 5.3: Results for LOC of The Proposed Technique

of these metrics as mentioned in the literature.

Table 5.1 and Figure 5.3 shows the LOC comparison between proposed technique and Rainbow. It is visible from the table that in all cases LOC is smaller in the proposed approach. For Monitor, Analyze, Plan, Execute, Knowledge Base Constructor (KBC) and Training, the proposed approach implementation contains 188, 25, 555, 528, 531 and 209 lines of codes respectively (Table 5.1 and Figure

Table 5.2: Descriptive Statistics for LCOM4 and MPC of The Proposed Method

Metric	*MinimumValue*	*MaximumValue*	μ	σ
LCOM4	0	3	0.8923	0.75256
MPC	0	17	3.046	4.40678



(a) LCOM Values of The Components (b) MPC Values of The Components

Figure 5.4: Results for LCOM4 and MPC of The Proposed Technique

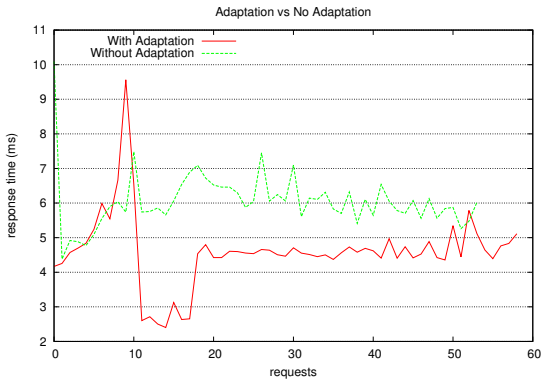
5.3a). For the same components, Rainbow contains 3694, 638, 1098, 3694 and 9124 lines of codes respectively. The improvement can be understood more clearly from Figure 5.3b. In case of Monitor and Analyze, the large difference is due to the use of utility functions to detect and analyze goal violations. While Rainbow uses complex model analysis for this purpose, the use of utility functions in the proposed approach simply compares the monitored values with the threshold for this. This is why it takes very few lines of code to implement this in the proposed method. For Plan and Execute, LOC is smaller because the adaptation logic is mathematical in nature rather than analytical like Rainbow where strategies need to be parsed and analyzed before execution. In total, the proposed technique has 2036 LOC in terms of all subcomponents while Rainbow has 9124 LOC (Figure 5.3c). Considering the full code base, the proposed method has 4367 LOC and Rainbow contains 24891 LOC (Figure 5.3c). The results prove that, in all cases, the proposed method has lower amount of LOC than Rainbow. So, the proposed method is more reusable than Rainbow from LOC measure.

Table 5.2 shows some descriptive statistics for LCOM4 and MPC of the proposed method which are minimum value, maximum value, mean and standard deviation. For LCOM4, the highest value is 3 and the lowest value is 0. The mean and standard deviation of this metric is 0.8923 and 0.75256 which indicates that LCOM4 values are close to the ideal values (0 and 1). The mean and standard deviation for MPC is 3.046 and 4.40678 which shows that, MPC values are low on average. This indicates low coupling between classes.

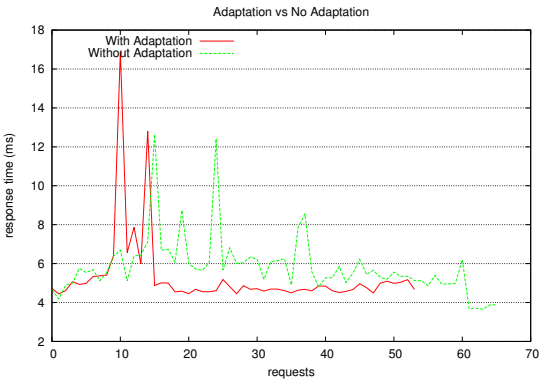
These results are more clearly visible from Figure 5.4. From Figure 5.4a, it is seen that 86.15% classes have LCOM4 values of either 0 or 1, where 13.85% classes have values different from these. This indicates that 86.15% classes achieved maximum cohesion. Figure 5.4 shows the number of classes for each of the MPC values. It is evident from the figure that most of the classes have low MPC values. This shows that the proposed methodology results in loosely coupled classes. So, according to the discussion in Section 5.4, low coupling and high cohesion show the reusability of the proposed technique.

The five runs of the adaptation logic for Znn.com is depicted in Figure 5.5. It is visible from all the five figures that adaptation improves the performance of the system gradually. The threshold chosen for performance was 6.2 milliseconds (ms). From Figure 5.5a, it is seen that response time gradually decreases after approximately 10 requests and becomes higher after approximately 18 requests. After this, the response time seems to become almost constant because of the constant load scenario as mentioned in Section 5.3. It is clear that when adaptation is applied, the response time gradually decreases under 6.2 ms and remains as such. In case of the design without adaptation, response time seem to be more frequently over the 6.2 ms from the figure.

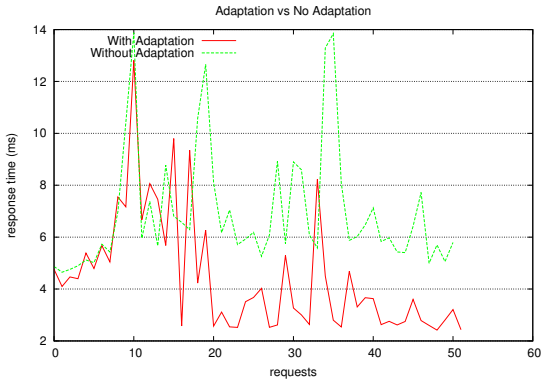
A similar pattern is seen from Figure 5.5b, the adaptation mechanism reduces the response time from the large spike after approximately 15 requests down to almost 5 ms. The system performs worse overall without adaptation because the



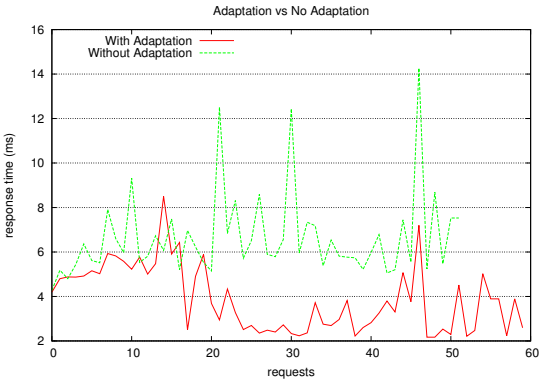
(a) Run 1



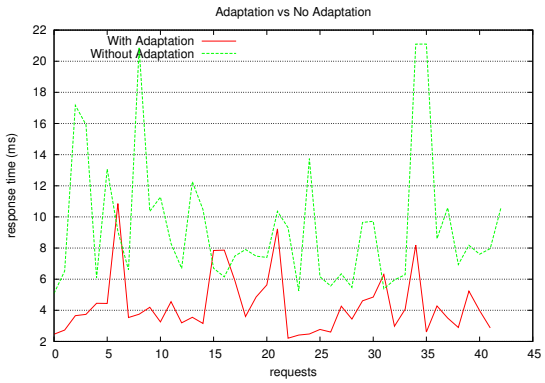
(b) Run 2



(c) Run 3



(d) Run 4



(e) Run 5

Figure 5.5: Comparison of Performance : Adaptation vs Without Adaptation in Five Runs

response time is higher than the threshold from 12 to 38 requests approximately.

From Figure 5.5c, the response time line also decreases after 15 requests. The response time becomes less than the threshold after approximately 15 requests and remains unchanged upto approximately 35th request when a sudden violation of response time goal is observed. However, it is evident that the response time

quickly drops back under the threshold and stays there throughout the run. From the figure, it is clear that adaptation steadily improves the response time in this scenario.

Figure 5.5d shows a similar pattern like Figure 5.5c. However, considering the previously mentioned four figures, it is clear that adaptation quality is gradually improved because the average distance between the response time lines with adaptation and without adaptation becomes distant. This happens due to the continuous update of knowledge base and training which ensures that the prediction model is up-to-date throughout the time.

Figure 5.5e represents the highest load run of all the five runs. In this case, the system becomes unstable and response time varies a lot. However, the adaptation mechanism still shows better performance than the system without any adaptation. In almost all cases, the mechanism without adaptation produces response time above the threshold where the system with adaptation crosses the threshold only five times, but runs down within threshold limit instantly.

From all the five figures mentioned previously, it is evident that the proposed adaptation scheme performs better than a system without adaptation in all cases. The LOC, MPC and LCOM4 values show that the proposed technique is able to achieve reuse in both component and subcomponent levels. So, from all the above discussion, it is observed that reuse is achieved while preserving effective adaptation. The adaptation mechanism has also been incorporated with a tool that generates the adaptation codes if inputs are given. So, it is expected that this will solve the reusability related research problem addressed in the literature [5, 60].

5.6 Summary

This chapter provides the result analysis for the proposed design mechanism for self-adaptive system. The goal was to achieve reusability while preserving the adaptation quality. In this chapter, a case study on a well-known system to the adaptive system community called Znn.com has been discussed. It has been seen that the proposed method has 4367 LOC in total where Rainbow, which used Znn.com for the first time has 24891 LOC in total. In case of subcomponent level LOC, the proposed technique has 2036 total subcomponent LOC while Rainbow has 9124 LOC. In terms of LCOM4, the mean and standard deviation of the approach is 0.8923 and 0.75256 respectively where 86.15% classes have ideal LCOM4 values (0 or 1). The MPC values of this method have 3.046 and 4.40678 mean and standard deviation respectively. From the analysis of the result, It is also noticeable that most of the modules have low MPC values. Thus, the values of these metrics show that reusability has been achieved. The adaptation technique has been compared to the situation with no adaptation in a high load scenario where it has achieved lower response times in all cases. On the whole, the proposed method improves the reusability of the model while maintaining the quality of adaptation.

Chapter 6

Conclusion

In this report, an adaptation mechanism assuring reusability and effectiveness has been proposed. The core contribution of this work is the development of a self-adaptive system design that not only enables reuse of the adaptation logic but also the reuse of adaptation logic components and subcomponents. This chapter summarizes the report and concludes by providing direction for future work.

6.1 Discussion

A reusable adaptation technique has been proposed in this report that can help to provide ready-made adaptation logic components to developers of self-adaptive systems. A tool has been developed to provide support for generating adaptation logic codes for a specific business logic component. The proposed adaptation component design has been tested on Znn.com, a well known model in the self-adaptive system community. It has been seen that the proposed methodology performs well in all cases.

The LOC measures for the proposed approach are 4367 and 2036 for whole code and subcomponent level respectively. For Rainbow, the LOC measures are 24891 and 9124 respectively for the same cases. It has been mentioned previously that LOC does not always represent the reusability of the code base and it has

been criticized in some literature [89, 4]. However, as the implementation for the proposed method and Rainbow were in the same language (Java) and the proposed technique outperforms Rainbow by a large amount of LOC, it can be concluded that it has achieved more reusability than the Rainbow framework.

The LCOM4 and MPC values represent cohesion and coupling. From the result analysis in Chapter 5, it has been seen that 86.15% classes have ideal LCOM4 values which is either 1 or 0. The MPC values have 3.046 and 4.40678 mean and standard deviation respectively. These results indicate that coupling is low but cohesion is high. This means that classes are mostly *single minded* and changing one class has low effect on other classes. So, it clearly shows that the adaptation design technique is reusable.

The 5 figures in Section 5.5 of Chapter 5 shows that adaptation mechanism always performs better than having no adaptation. The improvement could have occurred by chance if the experiments were not repeated. However, the experiments were repeated five times to ensure that the improvement that the adaptation technique shows is not merely by chance.

6.2 Threats to Validity

This section provides the threats that can hamper the accuracy and effectiveness of the proposed model. In total, three threats to validity have been identified which are given below.

1. The model assumes that accurate metric values can be achieved from the system for detecting and planning for adaptation. Without accurate metric values, the prediction model is inaccurate and so, the adaptation decision may be inaccurate as well.

2. The threshold for the metric needs to be carefully chosen. In case of selecting a threshold value that is unachievable, the adaptation logic cannot provide

an adaptation decision. It is mentionable that this is a common problem of the most self-adaptive system design methods [16, 12, 60].

3. The adaptation mechanism implementation depends on some third party libraries such as Weka, Commons IO etc. If any of these libraries contain inaccuracies, the adaptation decision is also affected.

4. It is assumed that the features and feature dependencies of the system are known, which is a valid assumption because development of a system occurs after requirement gathering, analysis and design. However, if these are not known for any reason, this adaptation design technique cannot be used.

6.3 Future Work

The proposed methodology provided a reusable adaptation component that can perform effective adaptation. However, the implementation is now available in only Java programming language. So, the tool also generates code in Java. In future, the implementation will be also provided in C#, PHP etc. languages. A more thorough case study will be performed on an industrial self-adaptive system to test the scalability of the approach.

The model will address the threshold problem discussed in the previous section in future. Currently, the model takes decision based on current situation only. In future, the technique will be enhanced to take adaptation decision by foreseeing future effects of the decision on the system.

Bibliography

[1] M. Luckey and G. Engels, "High-quality specification of self-adaptive software systems," in *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 143–152, IEEE Press, 2013.

[2] L. C. Briand, J. W. Daly, and J. K. Wust, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on software Engineering*, vol. 25, no. 1, pp. 91–121, 1999.

[3] M. Hitz and B. Montazeri, *Measuring coupling and cohesion in object-oriented systems*. Citeseer, 1995.

[4] 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*, pp. 2–8, ACM, 2006.

[5] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *Software engineering for self-adaptive systems*, pp. 48–70, Springer, 2009.

[6] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, p. 14, 2009.

[7] N. Abbas and J. Andersson, "Harnessing variability in product-lines of self-adaptive software systems," in *Proceedings of the 19th International Conference on Software Product Line*, pp. 191–200, ACM, 2015.

[8] R. Sterritt and D. W. Bustard, "Towards an autonomic computing environment," 2003.

[9] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[10] A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, *et al.*,

"Software engineering meets control theory," in *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 71–82, IEEE Press, 2015.

[11] 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*, pp. 76–85, IEEE, 2009.

[12] N. Esfahani, A. Elkhodary, and S. Malek, "A learning-based framework for engineering feature-oriented self-adaptive software systems," *Software Engineering, IEEE Transactions on*, vol. 39, no. 11, pp. 1467–1493, 2013.

[13] P.-C. David and T. Ledoux, "Towards a framework for self-adaptive component-based applications," in *Distributed Applications and Interoperable Systems*, pp. 1–14, Springer, 2003.

[14] T. Coupaye, É. Bruneton, and J. Stefani, "The fractal composition framework," *Specification, July*, 2002.

[15] 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 *Quality Software (QSIC), 2010 10th International Conference on*, pp. 288–293, IEEE, 2010.

[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*, pp. 7–16, ACM, 2010.

[17] B. S. Shang-Wen Cheng, "Model Problem: Znn.com." `https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/model-problem-znn-com/`.

[18] R. Laddaga and P. Robertson, "Self adaptive software: A position paper," in *SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems*, vol. 31, p. 19, Citeseer, 2004.

[19] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, pp. 1–26. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[20] A. K. Dey, "Understanding and using context," *Personal and ubiquitous computing*, vol. 5, no. 1, pp. 4–7, 2001.

[21] A. Computing *et al.*, "An architectural blueprint for autonomic computing," *IBM White Paper*, 2006.

[22] M. Salehie and L. Tahvildari, "Autonomic computing: emerging trends and open problems," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1–7, ACM, 2005.

[23] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, vol. 17, pp. 184–206, 2015.

[24] F. D. Macías-Escrivá, R. Haber, R. del Toro, and V. Hernandez, "Self-adaptive systems: A survey of current approaches, research challenges and applications," *Expert Systems with Applications*, vol. 40, no. 18, pp. 7267–7279, 2013.

[25] R. S. Group, "ROBOCUPRESCUE." `http://www.robocuprescue.org/`.

[26] "DEMANES." `http://www.demanes.eu/project_description`.

[27] M. Hüfner, S. Fischer, C. Sonntag, and S. Engell, "Integrated model-based support for the design of complex controlled systems," in *Symposium on Process Systems Engineering*, vol. 15, p. 19, 2012.

[28] S. Scholze, J. Barata, and O. Kotte, "Context awareness for self-adaptive and highly available production systems," in *Doctoral Conference on Computing, Electrical and Industrial Systems*, pp. 210–217, Springer, 2013.

[29] E. Sarriot and M. Kouletio, "Community health systems as complex adaptive systems: Ontology and praxis lessons from an urban health experience with demonstrated sustainability," *Systemic Practice and Action Research*, vol. 28, no. 3, pp. 255–272, 2015.

[30] V. Denneberg and P. Fromm, "Oscar. an open software concept for autonomous robots," in *Industrial Electronics Society, 1998. IECON'98. Proceedings of the 24th Annual Conference of the IEEE*, vol. 2, pp. 1192–1197, IEEE, 1998.

[31] J. Andersson, L. Baresi, N. Bencomo, R. de Lemos, A. Gorla, P. Inverardi, and T. Vogel, "Software engineering processes for self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II*, pp. 51–75, Springer, 2013.

[32] H. Müller, M. Pezzè, and M. Shaw, "Visibility of control in adaptive systems," in *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*, pp. 23–26, ACM, 2008.

[33] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, "Using architecture models for runtime adaptability," *IEEE software*, vol. 23, no. 2, pp. 62–70, 2006.

[34] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, no. 4, pp. 93–95, 2008.

[35] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg, "Models@ run.time to support dynamic adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, 2009.

[36] A. J. Ramirez and B. H. Cheng, "Design patterns for developing dynamically adaptive systems," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pp. 49–58, ACM, 2010.

[37] M. L. Berkane, L. Seinturier, and M. Boufaida, "Using variability modelling and design patterns for self-adaptive system engineering: application to smart-home," *International Journal of Web Engineering and Technology*, vol. 10, no. 1, pp. 65–93, 2015.

[38] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*, vol. 1. Prentice Hall Englewood Cliffs, 1996.

[39] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[40] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, and R. N. Taylor, "xadl: enabling architecture-centric tool integration with xml," in *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, pp. 9–pp, IEEE, 2001.

[41] J. Dowling and V. Cahill, "The k-component architecture meta-model for self-adaptive software," in *International Conference on Metalevel Architectures and Reflection*, pp. 81–88, Springer, 2001.

[42] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Future of Software Engineering, 2007. FOSE'07*, pp. 259–268, IEEE, 2007.

[43] "GME." `http://www.isis.vanderbilt.edu/projects/gme/`, 2008.

[44] "XTEAM." `http://softarch.usc.edu/~gedwards/xteam.html`, 2007.

[45] "Prism-MW." `http://sunset.usc.edu/~softarch/Prism/`, 2005.

[46] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, pp. 25–31, Feb. 2006.

[47] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[48] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plášil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski, "What characterizes a (software) component?," *Software-Concepts & Tools*, vol. 19, no. 1, pp. 49–56, 1998.

[49] M. R. Chaudron, C. Szyperski, and R. H. Reussner, *Component-based Software Engineering: 11th International Symposium, CBSE 2008, Karlsruhe, Germany, October 14-17, 2008, Proceedings*, vol. 5282. Springer, 2008.

[50] J. S. E. Bruneton, T. Coupaye, *The Fractal Component Model*. The ObjectWeb Consortium, Feb. 2004.

[51] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback control of computing systems*. John Wiley & Sons, 2004.

[52] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning*, vol. 6. Springer, 2013.

[53] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[54] H. N. Ho and E. Lee, "Model-based reinforcement learning approach for planning in self-adaptive software system," in *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*, p. 103, ACM, 2015.

[55] C. M. U. Software Engineering Institure, "Software Product Lines."

[56] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[57] R. Laddad, "I want my aop!, part 2," *Retrieved May*, vol. 11, p. 2004, 2002.

[58] P.-C. David and T. Ledoux, "An aspect-oriented approach for developing self-adaptive fractal components," in *International Conference on Software Composition*, pp. 82–97, Springer, 2006.

[59] M. Litoiu, M. Woodside, and T. Zheng, "Hierarchical model-based autonomic control of software systems," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1–7, ACM, 2005.

[60] N. Gui and V. De Florio, "Towards meta-adaptation support with reusable and composable adaptation components," in *Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference on*, pp. 49–58, IEEE, 2012.

[61] S.-W. Cheng, V. V. Poladian, D. Garlan, and B. Schmerl, "Improving architecture-based self-adaptation through resource prediction," in *Software Engineering for Self-Adaptive Systems*, pp. 71–88, Springer, 2009.

[62] J. Cámara, A. Lopes, D. Garlan, and B. Schmerl, "Adaptation impact and environment models for architecture-based self-adaptive systems," *Science of Computer Programming*, 2016.

[63] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2860–2875, 2012.

[64] J. Floch, C. Frà, R. Fricke, K. Geihs, M. Wagner, J. Lorenzo, E. Soladana, S. Mehlhase, N. Paspallis, H. Rahnama, *et al.*, "Playing musicbuilding context-aware and self-adaptive mobile applications," *Software: Practice and Experience*, vol. 43, no. 3, pp. 359–388, 2013.

[65] J. M. D. F. E. C. Rogue Wave, Patrick Thompson, "The CORBA Component Model (CCM)," Aug. 1999.

[66] "Introduction: Simulink Control." `http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=SimulinkControl`.

[67] "Robocode." `http://robocode.sourceforge.net/`, Dec. 2015.

[68] J. R. Quinlan *et al.*, "Learning with continuous classes," in *5th Australian joint conference on artificial intelligence*, vol. 92, pp. 343–348, Singapore, 1992.

[69] U. V. Sanjoy Dasgupta, Christos Papadimitriou, *Algorithms*. McGraw-Hill Science/Engineering/Math, 1 ed., 2006.

[70] G. Perrouin, B. Morin, F. Chauvel, F. Fleurey, J. Klein, Y. Le Traon, O. Barais, and J.-M. Jézéquel, "Towards flexible evolution of dynamically adaptive systems," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 1353–1356, IEEE Press, 2012.

[71] "Cas software." `http://www.cas.de/en/homepage.html`, Dec. 2015.

[72] "Fusion." `http://www.ics.uci.edu/~seal/projects/fusion/index.html`, June 2016.

[73] "Diva integrated studio." `https://sites.google.com/site/divawebsite/divastudio/diva-integrated-studio`.

[74] "Apache jmeter." `http://jmeter.apache.org/`.

[75] C. Kästner, S. Apel, M. Rosenmüller, D. Batory, G. Saake, *et al.*, "On the impact of the optional feature problem: analysis and case studies," in *Proceedings of the 13th International Software Product Line Conference*, pp. 181–190, Carnegie Mellon University, 2009.

[76] "Eclipse MARS." `https://eclipse.org/mars/`.

[77] "Weka." `http://www.cs.waikato.ac.nz/ml/weka/`.

[78] uklimaschewski, "EvalEx." `https://github.com/uklimaschewski/EvalEx`.

[79] "Commons IO." `http://commons.apache.org/proper/commons-io/`.

[80] "JUnit 4.12.0." https://github.com/junit-team/junit4.

[81] "mockito." http://site.mockito.org/.

[82] J. Cámara and R. de Lemos, "Evaluation of resilience in self-adaptive systems using probabilistic model-checking," in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 53–62, IEEE Press, 2012.

[83] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka, "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II*, pp. 76–107, Springer, 2013.

[84] M. Luckey, B. Nagel, C. Gerth, and G. Engels, "Adapt cases: extending use cases for adaptive systems," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 30–39, ACM, 2011.

[85] "slashdot.org." https://slashdot.org/.

[86] "Bash Reference Manual." https://tiswww.case.edu/php/chet/bash/bashref.html.

[87] J. Sztrik, "Basic queueing theory," *University of Debrecen, Faculty of Informatics*, vol. 193, 2012.

[88] S.-W. Cheng, *Rainbow: cost-effective software architecture-based self-adaptation*. ProQuest, 2008.

[89] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC Press, 2014.

[90] G. Gui and P. D. Scott, "Measuring software component reusability by coupling and cohesion metrics," *Journal of computers*, vol. 4, no. 9, pp. 797–805, 2009.

[91] D. A. Wheeler, "SLOCCount." http://www.dwheeler.com/sloccount/sloccount.html.

[92] "Cohesion metrics." http://www.aivosto.com/project/help/pm-oo-cohesion.html.