MULTILEVEL SPECIFICATION FOR ADAPTIVE SERVICES

A Thesis
Submitted to the Faculty
of
Purdue University
by
Phatak, Sucheta Chintamani

In Partial Fulfillment of the
Requirements for the Degree
of
Master Of Science

August 2009
Purdue University

Indianapolis, Indiana
DEDICATION

To Aai, Baba, Tai and Chaitanya
ACKNOWLEDGMENTS

I want to thank my parents and my sister Bhagyashri whose continual love and encouragement have been an essential ingredient in the development of this thesis. Their continuous questions regarding the status of my thesis and my graduate studies have always kept me focused and on track. I would also like to thank my husband, Chaitanya for his support and encouragement towards my education and my thesis. I would also like to thank my in-laws for believing in me and supporting me throughout my graduate course.

I would like to thank my professor and advisor, Dr. Rajeev R. Raje for giving me an opportunity to be a Research Assistant under him. His support and advice throughout the course of my graduate study and research work have helped me to successfully complete this thesis. His continuous feedback and motivation has helped me to thoroughly perform the research work. I am grateful for his constant encouragement, which made it possible for me to explore and learn new things. I would like to thank Dr. Snehasis Mukhopadhyay for being my co-advisor, and part of my Graduate Committee and providing valuable suggestions throughout the research period. I would also like to thank Dr. Xukai Zou for his guidance in my course work and for being on my thesis committee.

I would like to thank the entire staff of the Department of Computer and Information Science for their assistance during my graduate study. I would like to thank the IUPUI and Department of Computer and Information Science for the University Fellowship and the Teaching Assistantship support during my graduate study.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>x</td>
</tr>
<tr>
<td>ABBREVIATIONS</td>
<td>xi</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xii</td>
</tr>
<tr>
<td>CHAPTER 1. INTRODUCTION AND MOTIVATION</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Problem Definition</td>
<td>4</td>
</tr>
<tr>
<td>1.2. Goals &amp; Challenges</td>
<td>4</td>
</tr>
<tr>
<td>1.3. Overall Approach and Contributions</td>
<td>6</td>
</tr>
<tr>
<td>1.4. Organization</td>
<td>7</td>
</tr>
<tr>
<td>CHAPTER 2. RELATED WORK</td>
<td>8</td>
</tr>
<tr>
<td>2.1. Types of Adaptation</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1. Anticipated and Unanticipated Adaptations</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2. Survey of Practical Software Adaptation Techniques</td>
<td>9</td>
</tr>
<tr>
<td>2.2. Related work in Adaptive software systems</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1. Adaptive Middleware</td>
<td>10</td>
</tr>
<tr>
<td>2.2.2. Adaptive Frameworks</td>
<td>15</td>
</tr>
<tr>
<td>2.2.3. Adaptive SOA</td>
<td>19</td>
</tr>
<tr>
<td>2.2.4. Adapter Design pattern</td>
<td>20</td>
</tr>
<tr>
<td>2.3. Component Based Adaptation Techniques</td>
<td>21</td>
</tr>
<tr>
<td>2.3.1. A Formal Approach to Component Adaptation and Composition by David Hemer</td>
<td>21</td>
</tr>
<tr>
<td>2.3.2. A formal approach to component adaptation</td>
<td>22</td>
</tr>
<tr>
<td>2.3.3. Model-Based Adaptation of Behavioral Mismatching Components</td>
<td>24</td>
</tr>
<tr>
<td>2.3.4. Automating Component Adaptation for Reuse</td>
<td>25</td>
</tr>
<tr>
<td>2.3.5. Superimposition: A Component Adaptation Technique</td>
<td>25</td>
</tr>
<tr>
<td>2.3.6. Module Reuse by Interface Adaptation</td>
<td>26</td>
</tr>
<tr>
<td>CHAPTER 3. Design and implementation</td>
<td>28</td>
</tr>
<tr>
<td>3.1. Definition of Adaptive Service</td>
<td>28</td>
</tr>
<tr>
<td>3.2. UMM Specification</td>
<td>29</td>
</tr>
<tr>
<td>3.3. Multi-level Specification of an Adaptive Service</td>
<td>32</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>----</td>
</tr>
<tr>
<td>3.3.1. Information Classifier Service</td>
<td>32</td>
</tr>
<tr>
<td>3.3.2. Design of Specification for Adaptive Service</td>
<td>35</td>
</tr>
<tr>
<td>CHAPTER 4. IMPLEMENTATION</td>
<td>74</td>
</tr>
<tr>
<td>4.1. Service Attributes</td>
<td>74</td>
</tr>
<tr>
<td>4.2. Syntax Level Mapping</td>
<td>75</td>
</tr>
<tr>
<td>4.2.1. Adaptation in order and number of arguments</td>
<td>76</td>
</tr>
<tr>
<td>4.2.2. Type Adaptation</td>
<td>80</td>
</tr>
<tr>
<td>4.2.3. Name Adaptation</td>
<td>83</td>
</tr>
<tr>
<td>4.3. Semantic Level Mapping</td>
<td>84</td>
</tr>
<tr>
<td>4.3.1. Adaptive PreCondition</td>
<td>86</td>
</tr>
<tr>
<td>4.3.2. Adaptive PostCondition</td>
<td>88</td>
</tr>
<tr>
<td>4.4. Synchronization Level Mapping</td>
<td>90</td>
</tr>
<tr>
<td>4.5. QoS Mapping</td>
<td>97</td>
</tr>
<tr>
<td>4.6. Mapping Learning Functions</td>
<td>102</td>
</tr>
<tr>
<td>4.7. Order of Mapping</td>
<td>103</td>
</tr>
<tr>
<td>CHAPTER 5. TRANSLATOR IMPLEMENTATION AND CASE STUDY</td>
<td>112</td>
</tr>
<tr>
<td>5.1. Design of TraMS</td>
<td>112</td>
</tr>
<tr>
<td>5.1.1. GUI</td>
<td>114</td>
</tr>
<tr>
<td>5.1.2. MS_Parser</td>
<td>115</td>
</tr>
<tr>
<td>5.1.3. MS_Translator</td>
<td>116</td>
</tr>
<tr>
<td>5.2. Case Studies</td>
<td>118</td>
</tr>
<tr>
<td>5.2.1. Adaptive Classifier Service</td>
<td>118</td>
</tr>
<tr>
<td>5.2.2. Adaptive Container Service</td>
<td>129</td>
</tr>
<tr>
<td>5.2.3. Adaptive Tracking Service</td>
<td>130</td>
</tr>
<tr>
<td>5.3. Empirical Evaluation for Adaptive Classifier</td>
<td>130</td>
</tr>
<tr>
<td>5.3.1. Plain Classifier Service</td>
<td>131</td>
</tr>
<tr>
<td>5.3.2. Adaptive Classifier</td>
<td>131</td>
</tr>
<tr>
<td>5.3.3. Empirical Evaluation</td>
<td>131</td>
</tr>
<tr>
<td>CHAPTER 6. CONCLUSION AND FUTURE WORK</td>
<td>141</td>
</tr>
<tr>
<td>6.1. Conclusions</td>
<td>141</td>
</tr>
<tr>
<td>6.2. Contributions</td>
<td>142</td>
</tr>
<tr>
<td>6.3. Future Work</td>
<td>143</td>
</tr>
<tr>
<td>6.4. Summary</td>
<td>143</td>
</tr>
</tbody>
</table>
LIST OF REFERENCES ................................................................. 145
APPENDIX ................................................................. 151
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.1 Classify function of Classifier Service</td>
<td>1</td>
</tr>
<tr>
<td>Table 1.2: Classify function of Classifier Service returning String</td>
<td>2</td>
</tr>
<tr>
<td>Table 2.1: Comparison between unanticipated and anticipated adaptations</td>
<td>9</td>
</tr>
<tr>
<td>Table 3.1: Signature of classify function of the classifier service</td>
<td>33</td>
</tr>
<tr>
<td>Table 3.2: UMM specification of the classifier service</td>
<td>33</td>
</tr>
<tr>
<td>Table 3.3: Syntax specification of a function in XML format</td>
<td>36</td>
</tr>
<tr>
<td>Table 3.4: Syntax specification of the classify function in XML format</td>
<td>36</td>
</tr>
<tr>
<td>Table 3.5: Syntax specification of a function supporting adaptive names</td>
<td>38</td>
</tr>
<tr>
<td>Table 3.6: Syntax specification of the classify function supporting adaptive names</td>
<td>38</td>
</tr>
<tr>
<td>Table 3.7: Syntax specification of a function supporting Type Adaptation</td>
<td>40</td>
</tr>
<tr>
<td>Table 3.8: Syntax specification of the classify function supporting Type Adaptation</td>
<td>40</td>
</tr>
<tr>
<td>Table 3.9: Syntax specification of a function supporting order and number adaptation</td>
<td>41</td>
</tr>
<tr>
<td>Table 3.10: Syntax specification of the classify function supporting order and number adaptation</td>
<td>44</td>
</tr>
<tr>
<td>Table 3.11: Semantic specification of a function</td>
<td>45</td>
</tr>
<tr>
<td>Table 3.12: Semantic specification of the classify function</td>
<td>46</td>
</tr>
<tr>
<td>Table 3.13: Semantic specification of a function supporting adaptive pre-condition</td>
<td>47</td>
</tr>
<tr>
<td>Table 3.14: Semantic specification of the classify function which supports adaptive pre-condition</td>
<td>48</td>
</tr>
<tr>
<td>Table 3.15: Semantic specification of a function supporting adaptive post-condition</td>
<td>49</td>
</tr>
<tr>
<td>Table 3.16: Semantic specification of the classify function which supports adaptive post-condition</td>
<td>50</td>
</tr>
<tr>
<td>Table 3.17: Synchronization specification of a function</td>
<td>52</td>
</tr>
<tr>
<td>Table 3.18: Synchronization specification of the classify function</td>
<td>52</td>
</tr>
<tr>
<td>Table 3.19: Synchronization specification of a function supporting synchronization policy adaptation</td>
<td>54</td>
</tr>
<tr>
<td>Table 3.20: Relation between Mutual Exclusion policy and FCFS policy</td>
<td>55</td>
</tr>
<tr>
<td>Table 3.21: Synchronization specification of the classify function supporting synchronization policy adaptation</td>
<td>55</td>
</tr>
<tr>
<td>Table 3.22: Synchronization specification of a function supporting synchronization technique adaptation</td>
<td>56</td>
</tr>
</tbody>
</table>
Table 3.23: Synchronization specification of the classify function supporting synchronization technique adaptation ................................................................. 57
Table 3.24: QoS specification of a function .......................................................... 58
Table 3.25: QoS specification of the classify function ............................................. 58
Table 3.26: QoS specification of an adaptive function ........................................... 60
Table 3.27: QoS specification of the adaptive classify function .............................. 60
Table 3.28: QoS specification of a function implementing a learning technique .. 62
Table 3.29: QoS specification of the classify function implementing reinforcement learning technique ................................................................. 63
Table 3.30: QoS specification indicating learning properties of a function implementing a learning technique ................................................................. 65
Table 3.31: QoS specification indicating learning properties of the classify function implementing reinforcement learning technique ................................................................. 66
Table 3.32: Specification for feedback function associated with a learning technique ................................................................................................................. 67
Table 3.33: Specification for init function associated with a learning technique . 67
Table 3.34: Specification for feedback function associated with the classify function ................................................................................................................. 68
Table 3.35: Complete Multi-level specification of the classifier service ............... 68
Table 4.1: Comment block for the Inherent Attributes of the classifier service ... 75
Table 4.2: Core-function skeleton for classify function of the classifier service .. 76
Table 4.3: Wrapper function skeleton for classify function adapting in order of arguments ................................................................................................................. 77
Table 4.4: Dummy function signature with some default arguments .................... 78
Table 4.5: Main wrapper function for Dummy function ........................................... 78
Table 4.6: Overloaded versions of wrapper function .............................................. 78
Table 4.7: Wrapper functions skeleton for classify function adapting in number of arguments ................................................................................................................. 79
Table 4.8: MSAS of the classify function supporting adaptive return type .......... 81
Table 4.9: Wrapper function for the classify function supporting adaptive return type ................................................................................................................. 81
Table 4.10: Wrapper function for the classify function supporting adaptive argument type ................................................................................................................. 83
Table 4.11: Dummy functions for the classify function supporting adaptive names ................................................................................................................. 84
Table 4.12: Core-function skeleton for classify function of the classifier service with assertions added by TraMS ................................................................. 86
Table 4.13: Code added to skeleton of wrapper function to check adaptive pre-conditions ................................................................................................................. 86
Table 4.14: Wrapper function skeleton for classify function of the classifier service with conditional checks added by TraMS for adaptive pre-conditions .... 87
Table 4.15: Code added to skeleton of wrapper function to check adaptive post-conditions ................................................................................................................. 87
Table 4.16: Wrapper function skeleton for classify function of the classifier service with conditional checks added by TraMS for adaptive post-conditions .. 90
Table 4.17: *init* function skeleton for a function supporting Synchronization policy adaptation.................................................................................................................. 94
Table 4.18: Skeleton code generated by TraMS for classify function of classifier service supporting synchronization policy adaptation .......................................................... 95
Table 4.19: QoS specification for classify function’s static parameter .................. 97
Table 4.20: QoS specification mapping for classify function’s static parameters 98
Table 4.21: Dynamic QoS parameter specification for classify function .......... 100
Table 4.22: Dynamic QoS parameter mapping for classify function ............... 101
Table 4.23: Skeleton code for getter function of Learnt Parameter ............... 102
Table 4.24: Skeleton code for feedback and init function ............................. 103
Table 4.25: Stub generated by TraMS for Adaptive Classifier Service ........... 105
Table 5.1: MSAS (XML) of the Adaptive classifier service ............................ 118
Table 5.2: Stub generated by TraMS for Adaptive Classifier Service ......... 123
Table 5.3: Disk Footprint and LOC analysis ................................................. 132
Table 5.4: Turn Around time required for classifying 10 different documents by Non-adaptive classifier and Adaptive classifier with no adaptations ................. 133
Table 5.5: Turn-around-time required for different adaptations by the adaptive classifier service .................................................................................................................. 134
Table 5.6: Turn-around-time for adaptive service with multiple adaptations..... 136
Appendix Table
Table A.1: MSAS (XML) for Adaptive Container Service ......................... 151
Table A.2: Java stub generated for MS(XML) of Adaptive Container Service. 155
Table A.3: MSAS (XML) for Adaptive Tracking Service ............................ 161
Table A.4: Java stub generated for MSAS (XML) of Adaptive Tracking Service ................................................................................................................................. 163
Table A.5: XML Specification for MSAS .................................................... 166
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Feature diagram for multi-level service specification.</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>Feature diagram for Inherent Attributes of a service.</td>
<td>31</td>
</tr>
<tr>
<td>3.3</td>
<td>Feature diagram for Functional Attributes of a service.</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Feature diagram for Function and Contract of a service.</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>Order for mapping different adaptation types</td>
<td>105</td>
</tr>
<tr>
<td>5.1</td>
<td>Structural Design of TraMS</td>
<td>113</td>
</tr>
<tr>
<td>5.2</td>
<td>Class Diagram for TraMS</td>
<td>114</td>
</tr>
<tr>
<td>5.3</td>
<td>Screen shot for GUI of TraMS</td>
<td>115</td>
</tr>
<tr>
<td>5.4</td>
<td>Class Diagram for intermediate service representation</td>
<td>117</td>
</tr>
<tr>
<td>5.5</td>
<td>Classification success variations for Plain Service</td>
<td>138</td>
</tr>
<tr>
<td>5.6</td>
<td>Classification success variations for Adaptive Service</td>
<td>139</td>
</tr>
<tr>
<td>Appendix</td>
<td>TraMS GUI Screenshot 1</td>
<td>176</td>
</tr>
<tr>
<td>A.2</td>
<td>TraMS GUI Screenshot 2</td>
<td>177</td>
</tr>
<tr>
<td>A.3</td>
<td>TraMS GUI Screenshot 3</td>
<td>178</td>
</tr>
<tr>
<td>A.4</td>
<td>TraMS GUI Screenshot 4</td>
<td>179</td>
</tr>
<tr>
<td>A.5</td>
<td>TraMS GUI Screenshot 5</td>
<td>180</td>
</tr>
<tr>
<td>A.6</td>
<td>TraMS GUI Screenshot 6</td>
<td>181</td>
</tr>
</tbody>
</table>
ABBREVIATIONS


TraMS: Translator for Multi-level Specification.

SOA: Service Oriented Architecture.

The words service and component are being used interchangeably in this thesis.

The words interface and specification are used interchangeably in this thesis.
ABSTRACT


Software services which are integral part of a distributed system are traditionally developed using a fixed set of design decisions. Such a set places limits on the capabilities of the service. Factors such as changes in the execution environment, the business context, and client requirements demand that services be made adaptive in their behavior. Change being the only constant parameter in any software system, a service developer needs to consider adaptation right from the design phase instead of treating it as an ad-hoc attribute.

The challenge in creating adaptive services not only lies in designing and implementing such services but also in describing these services using an associated formal specification or contract. This thesis presents an approach to create adaptive services considering adaptation at different levels (syntax, semantics, synchronization and QoS) of service contracts. A formal grammar for specifying adaptive services is proposed, which helps a service developer to create adaptive specification of their services. This specification can be then registered with any discovery system which matches client queries with service specifications. Also, the mapping of this adaptive multi-level specification onto a specific programming language, such as Java, is proposed. Thesis also evaluates a case study of a distributed information classifier service. It also presents an empirical evaluation of effect of adaptations on turn-around-time, accuracy, disk size, LOC of the classifier service.
CHAPTER 1. INTRODUCTION AND MOTIVATION

A typical component-based distributed system consists of independently developed and deployed services, which can be directly used by clients to perform specific functionalities. Throughout their lifetime, these services may be faced with variety of changes. These changes can be classified into external changes and internal changes. External changes include a trigger from outside the service, such as the environmental changes or context changes. The environment, in which the service is hosted, may suffer changes in resource availability such as network, memory, and devices. Also, a service may experience an internal event such as performance degradation which may demand changes in its behavior. All such external or internal events may necessitate an adaptive behavior. Addressing such adaptations has to be considered while designing and implementing the software component and not as an afterthought. This mechanism allows the service developer to “design for change” making the service adaptable or adjustable to changes.

Adaptation in software services can be triggered due to following stimuli:

- Change in client’s requirements

Consider a Distributed Classifier service, providing following function, typically used in information filtering systems [RAJ97].

<table>
<thead>
<tr>
<th>Table 1.1 Classify function of Classifier Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public DocumentClass Classify(Document doc)</td>
</tr>
<tr>
<td>requires doc!=null</td>
</tr>
<tr>
<td>ensures return!=null</td>
</tr>
</tbody>
</table>
This distributed classifier service provides a function called “Classify”. This function accepts a document which the client wants to classify. It also has a precondition that the given document cannot be null. The function also ensures that the returned DocumentClass of the document is not null. A client of this service may want a classifier service with returned parameter String representing the DocumentClass. Thus, a similar classifier service is needed but with the “classify” function now changed to:

```
Table 1.2: Classify function of Classifier Service returning String

<table>
<thead>
<tr>
<th>public String Classify(Document doc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>requires doc! = null</td>
</tr>
<tr>
<td>ensures return! = EmptyString</td>
</tr>
</tbody>
</table>
```

Above example indicates adaptation of a service due to change in the client’s requirement.

- Change in the execution environment

In a dynamic execution environment that hosts a service, it may be inappropriate to stop and restart the service to handle the environment changes such as network failure, device failures, device quality degradation, resource inavailability. Such changes could be frequent or erratic, so, the adaptations must be applied by the service dynamically. Some of these changes such as device failures, device quality degradation, can be foreseen prior to the development of the service by the service developer. In that case, the service can be implemented taking into consideration the required adaptations. For instance, a distributed tracking service which uses optical cameras to track objects can experience changes in its environment. At run-time the optical camera devices may fail or start degrading in quality. In this case, the tracking service should adapt to handle these changes in such a way that it satisfies the service contract.

- Change in the internal state of the service
A composite service in a distributed system can be assembled from various other cooperating services. These sub-services can have a specific communication pattern. Anomalies in this inter-service communication pattern may require the service to adapt to handle these changes. For instance, consider a distributed classifier service which cooperates with other classifiers in the system to provide successful classification. At run-time one of the cooperating "acquaintances" may leave the system or new efficient classifiers may join the system. In these cases, the classifier service should adapt its acquaintances dynamically, to satisfy its contracts. Considering these acquaintances to be integral part of the classifier service, there is a need to adapt to internal state changes of the service.

An adaptive service may address any or all of these stimuli. If a service is adaptive then its interface needs to indicate the adaptive feature explicitly. Such an explicit indication will serve two purposes: a) it can act as a guide for the developer while creating the service and b) a client can be made aware of the adaptive nature of the service and it can take advantage of the adaptive features by making appropriate calls. Also, such an adaptive specification may enable a client to make an appropriate selection of a service from the available choices.

Beugnard et al. [BEU99], have identified four classes of contracts that specify a service: a) basic or syntactic, b) behavioral, c) synchronization, and d) quantitative.

Such multi-level contracts serve many purposes such as boosting the client’s confidence about a service and help the service developer in better understanding the requirements of an application. Similarly, when an adaptive service is to be developed, the service developer has to indicate the adaptive characteristics as part of the service specification. This thesis proposes that any adaptive service can completely describe its adaptive behavior using an enhancement of the four-level contract suggested by Beugnard et al. [BEU99]. This enhancement is achieved by proposing a formal mechanism for specifying adaptation at syntax, behavior, synchronization, and quantitative levels. The proposed enhanced specifications will be used by the developers for designing
and implementing adaptive services. This formalized adaptive interface will also help the clients in following ways:

- Clients can identify adaptive services from non-adaptive services. Thus, if their application domain demands adaptation they can choose the adaptive service possibly with a performance penalty.
- If at all clients choose to integrate the adaptive service, from the specification they will understand different adaptations it supports at all the levels of contract.

Along with the formal specification, this thesis proposes a translator for converting this specification into Java skeleton code. This translator will assist the service developer in implementing the adaptive service by automating the Java skeleton code generation.

Conclusively, a well designed formal specification of an adaptive service will serve as a blueprint of the service and assist the development of adaptive services.

1.1. Problem Definition

As discussed earlier, there is a need of designing a multi-level specification for adaptive services. Hence, this thesis aims to design and develop a Multi-level Specification for Adaptive Services (MSAS) and a Translator for this specification, which facilitates the development of adaptive services by creating Java skeleton code.

1.2. Goals & Challenges

The specific objectives of the thesis and associated challenges are:

- To create a Multi-level Specification for Adaptive Services (MSAS) and a Translator for this Multi-level Specification (TraMS) that involves:
  - Designing a generic multi-level interface. This interface describes an adaptive service at the syntax, semantics, synchronization and QoS levels.
The challenge here is to first identify different types of generic adaptations a service can employ and how these adaptations map to different levels of contract. This involves understanding the types of adaptations and underlying theory as well as their interrelation with syntax, semantics, synchronization and QoS of the service. Next challenge is to formally and concisely represent these adaptations in the interface. This interface should indicate all necessary types of adaptations that clients would be interested in, while being readable and understandable.

- Identifying the mapping between such an interface and popularly used object oriented language constructs. This mapping will be used in the TraMS that generates skeleton code from a given multi-level interface.

This objective demands the understanding of the implementation aspects of the adaptive service. This involves identifying the mapping between different adaptations indicated in the adaptive interface and an object oriented language such as Java. While doing so, correct alternatives should be chosen from available implementations.

- To empirically validate the usage of MSAS and TraMS considering a comprehensive case study.

- The challenge here is to identify the pros and cons of the MSAS and TraMS using a moderately complex real world software scenario. Various metrics such as ease of use, size of additional code generated and associated performance penalties will be used in the validation process.
1.3. **Overall Approach and Contributions**

Considering the challenges demanded in implementing the MSAS, an iterative approach is taken to solve the problem. Initially, it involves understanding different types of adaptations possible in distributed services and their relation with different levels of contract. A more difficult challenge is to understand efficient way of specifying these adaptations in the interface. This involved amending existing techniques for contract specification to indicate adaptive behavior. Different interface constructs are created so as to facilitate the representation of adaptation at every level of contract concisely.

Once the structure for the MSAS is identified, associated grammar and schema for the interface is created. The grammar and schema assist in the process of implementing the TraMS for the specification. Following points are considered while developing this interface:

- The interface is generic and can represent an adaptive service in any application domain.
- The interface is not tied to any specific programming language. The constructs used in the interface can be translated to any well-known object oriented programming language.

A translator is created based on the relation between adaptive interface and an object oriented language. This translator transforms the adaptive interface into JAVA by creating adaptation stubs which are later modified by the service developer.

To demonstrate the practical application of this translator a case study is considered. The case study involves creating an adaptive service interface for a real life service and then using the TraMS to create the implementation stubs. Actual implementation code is then inserted in these stubs to create an adaptive service.
1.4. Organization

This thesis is organized into six chapters. An Introduction, along with the problem definition, goals and challenges, overall approach and contributions is provided in this chapter. Chapter 2 discusses the background and the related work on this thesis. Chapter 3 describes the design details of the MSAS. Chapter 4 provides the mapping details for MSAS to Java language. Chapter 5 provides case-study to validate and verify the usage of MSAS and TraMS. Chapter 6 provides the conclusion of this research work and the possible future extensions.
CHAPTER 2. RELATED WORK

This chapter discusses the work done in regards to software service adaptation and its representation in the service specification. Following different domains are identified while considering the related work.

- Types of adaptation
- Related work in Adaptive software systems.
  - Adaptive Middleware
  - Adaptive Frameworks
  - Adaptive SOA
  - Adapter patterns
- Component Based Adaptation Techniques

2.1. Types of Adaptation

2.1.1. Anticipated and Unanticipated Adaptations
The paper [CREM 06] by Cremene et al., defines two different types of adaptations Anticipated and Unanticipated. Anticipated adaptation involves predicting possible changes in the service context along with specifying rules or actions for each such predicted situation. These anticipated adaptations are customizations which provide a set of options from which client can choose. In unanticipated or dynamic adaptation, the service evolves as the context evolves. Unanticipated adaptation deals with handling changes that the service developer did not anticipate while designing it.
Following table identifies the key difference between these two types of adaptations:

<table>
<thead>
<tr>
<th>Anticipated Adaptations</th>
<th>Unanticipated Adaptations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predetermined adaptation strategies.</td>
<td>Best strategy is searched.</td>
</tr>
<tr>
<td>Service architecture can deal with any anticipated context.</td>
<td>Service architecture evolve only if the context changes.</td>
</tr>
<tr>
<td>Architecture has to be redeployed if the context evolves.</td>
<td>Context evolution is possible while the service is running.</td>
</tr>
<tr>
<td>Service developer needs to anticipate and specify the rules and strategies for each adaptive service.</td>
<td>Service developer’s role is simplified. The service is adaptive and it evolves.</td>
</tr>
</tbody>
</table>

This thesis considers only anticipated adaptations while developing the generic adaptive interface for a software service. This is because the service developer has to predetermine the adaptations supported by the service and needs to specify these strategies in the specification. To create a MSAS, the Service developer has to predetermine possible adaptations for a service, but can dynamically decide the way to implement these adaptations. Thus in MSAS, all possible adaptations have to be anticipated but their implementations can be unanticipated.

2.1.2. Survey of Practical Software Adaptation Techniques
The paper [Kell08] by Stephen Kell defines adaptation and surveys the related research work in the area of software adaptations. The paper describes and contrasts the approaches of each of the research work. It defines adaptation as
“any process which modifies or extends the implementation or behavior of a subsystem to enable or improve its interactions, or synonymously, its communication with the surrounding parts of the system (which we call its environment).”

Then the paper distinguishes multiple kinds of adaptations. First is adaptation done for “Functionality and Correctness”, which is the ability for two subsystems to communicate in a way which preserves the “meaning”. The second is adaptation for extra-functional properties such as Quality of Service. Next is the adaptation in the form of the ability of a system to configure itself in response to changing environment like, network conditions, resource constrains. Finally, the adaptation related to dynamically discovering new components or devices and automatically constructing an adaptor which allows them to successfully communicate.

This thesis considers an extended definition of adaptation based on Kell’s definition. The thesis deals with only those types of adaptations which directly or indirectly affect the contract of the service with the outside world.

2.2. Related work in Adaptive software systems
Adaptive software component changes its behavior at runtime. This special class of software has a number of potential benefits, ranging from the ability to respond rapidly to security threats to the opportunity to optimize performance as characteristics of the underlying execution environment change. This section describes the techniques suggested by various researchers to make adaptations possible.

2.2.1. Adaptive Middleware
Not only the Inherent dynamics but the heterogeneity of component technologies and network protocols make distributed applications more difficult to develop and manage. These difficulties are drastically overcome by Middleware technologies
which provide a layer of abstraction. Traditional middleware (such as CORBA, DCOM, and Java RMI) addresses most of these problems to some extent through the use of a “black-box” approach, such as encapsulation in object-oriented programming. However, traditional middleware is limited in its ability to support adaptation. To address this issue, adaptive middleware has evolved from traditional middleware. In addition to the object-oriented programming paradigm, adaptive middleware employs several other key technologies including computational reflection, component-based design, aspect-oriented programming, and software design patterns. Adaptive characteristics in a software system can either be part of the middleware or of the component itself. Following are the examples of adaptive middleware efforts. These following adaptive middleware enforce adaptation on the services that use the middleware. According to end to end argument [SALT 84], only the most basic functionalities should be considered as part of the middleware and all other fancy features should be left for the service to develop. A service which does not want to be adaptive due to real time constraints or for avoiding adaptation overhead is thus free to leave adaptive implementation. This approach is taken by this thesis. Also in some cases, making the middleware adaptive need not make the service adaptive. Adaptation at service level can be totally different that adaptation at middleware. Thus, along with the middleware all the services which require adaptation have to be designed considering it. Hence, unlike these adaptive middleware technologies this thesis tries to enforce “design by change” in services considering adaptive features. This view enables considering middleware as yet another service which abstracts heterogeneity to be designed as adaptive service.

2.2.1.1. RAPIDware
The RAPIDware [RAPID01] project proposes the design and use of adaptive, component-based middleware in dynamic, heterogeneous environments. The project addresses ways of enhancing existing applications with new adaptive
behavior. The primary focus is on approaches that can be realized through generative programming techniques. The project has investigated design technologies like contract-based software design, safe adaptation, and analysis and verification of adaptive software. Project considers the use of machine learning techniques to support decision-making for adaptation. Collectively, these techniques comprise a new programming model, called transparent shaping. Without modifying the application source code, these techniques enable new adaptive behavior to be woven into existing programs. This integration involves aspect-oriented programming and computational reflection. RAPIDware is involved in designing and implementing transparent shaping tools for commonly used programming languages and middleware platforms. Similar to the approach in this thesis, RapidWare project considers contract based design to provide guarantee in adaptive software. They have introduced TA-LTL, a timed adaptation based extension to linear temporal logic. This logic is used to specify three timing properties associated with the adaptation process: safety, liveness, and stability.

2.2.1.2. DynamicTAO
DynamicTAO [DYNTAO 99] is an extension of the TAO ORB. The TAO ORB allows different aspects of its operation to be selected at load time using the Strategy design pattern. These aspects are specified in a configuration file. DynamicTAO allows these strategies to be inspected and changed at runtime in a reflective manner, while still maintaining consistency. This is achieved via the DynamicConfigurator component that supports dynamic querying of the current ORB strategies and dynamic uploading of new strategies. However, only a fixed number of locations within the ORB can be adapted. Strategy implementations can also be uploaded to or downloaded from a remote location, then selected for use. A runtime graphical interface, Doctor (Dynamic ORB Configuration Tool), is introduced to support this possibly remote runtime dynamic adaptation. The DomainConfigurator, TAOConfigurator, and ServantConfigurator are all
realizations of service configurator pattern in DynamicTAO. A service configurator in DynamicTAO exports the DynamicConfigurator interface, which is a CORBA IDL interface, defined also as the MOP (Meta-Object Protocol) for inspecting, adapting, loading, and unloading “component implementations” dynamically. Thus considering a middleware as a service, DynamicTAO is an adaptive version of this service. On the other hand this thesis concentrates on providing a multi-level specification for such adaptive services or components.

2.2.1.3. OpenORBv, OpenORB v2

In OpenORB project, the successor of the Adapt project, Blair et al. [BLAIR 98] continued their investigation studying the role of computational reflection in middleware. The design of OpenORB uses the concept of reflection to provide the desired level of configurability and openness. It includes the ability to associate a metaspace with every object/interface. This metaspace is subdivided into three models, encapsulation, composition and environment. The project considers the use of object graphs to represent composite components in the architecture. The architecture of OpenORB provides a language-independent model of reflection.

Subsequently, Blair et al. designed OpenORB v2 [BLAIR01] that adds a component-based design framework to the OpenORB reflective framework. An instance of OpenORB v2 is a particular configuration of components that can be selected at build and reconfigured at runtime. In reflective systems, structural reflection deals with the content and structure of a given component. In OpenORB v2 architecture, this aspect of metaspace is represented by two distinct metamodels, namely the interface and architecture metamodels. The two metamodels represent a separation of concerns between the external view of a component (its set of interfaces) and the internal construction (its software architecture). Metaspace model for behavioral reflection has been classified in interception and resources metamodels. The interception metamodel enables the
dynamic insertion of interceptors. These interceptors are associated with interfaces and enable the insertion of pre- and post-behavior. The resources metamodel provides access to underlying resources and resource management. The project has developed an enhanced architectural description language (ADL) called Xelha[CAZ99], which builds on their task model. They have implemented a tool for the interpretation of Xelha specifications. Xelha supports the specification of software architectures in terms of components, their interfaces, and connectors. Thus Xelha is an architectural specification unlike MSAS, a service specification.

Thus the OpenORB, along with other features, provides the way to achieve insertion of pre and post-behavior to an interface, while on the other hand MSAS provides way to represent such pre and post-behaviors.

2.2.1.4. AspectIX

AspectIX [GEI98] is a new middleware architecture that extends CORBA and remains fully compliant with CORBA. It uses a model of distributed objects consisting of several fragments instead of a client-server model. It uses aspects to specify nonfunctional properties like replication strategies, consistency models, etc. in an application. This results in dynamic and transparent exchanges of implementations.

The AspectIX architecture adopts a fragmented object model. A distributed object consists of several fragments, which can interact with each other. When a fragment is created, e.g., as a result parameter of a method invocation, the ORB creates two local objects in the desired target language: a fragment interface and a fragment implementation. The fragment interface is a generic object that is automatically generated during the development process. Its main purpose is to delegate method calls to the fragment implementation. Nonfunctional aspects can be specified via a typed configuration interface. This interface determines the local fragment implementation that represents the distributed object. When the
aspect specification changes at runtime and cannot be satisfied by the currently active implementation, it is transparently exchanged by another one.

Thus AspectIX provides the way of achieving runtime adaptation using aspects, on the other hand, MSAS represents what type of adaptations are performed by a service using a formal specification. Thus adaptations represented in a MSAS can be implemented using techniques proposed in AspectIX.

2.2.2. Adaptive Frameworks

Diverse research has been performed in the area of adaptive frameworks, which provide necessary toolsets to create adaptive services. This section presents some of these frameworks used for adaptive software development. Comparing these frameworks with the work in this thesis, the thesis considers adaptive behavior from the design of the component and not as an afterthought. Also, this thesis addresses the issues related to specification of an adaptive service and not the implementation.

2.2.2.1. Framework for Self-adaptive Component-Based Applications

David et al., [DAV03] propose an approach for adaptation based on the Separation of Concerns. They propose “adaptation to a specific execution context and its evolutions as a concern which should be treated separately from the rest of an application”. They propose the addition of the adaptation logic, to non-adaptive code. The adaptation logic deals specifically with the adaptation concern. This results in a self-adaptive application which reconfigures its architecture and parameters to always match its changing environment. They present the use of their development framework and tools which is based on Fractal component model [COUP 02]. This framework supports two different kinds of reconfigurations, structural and parameterization. It uses reflection to provide meta-programming interface to Fractal components. Adaptation logic is
added into an application using adaptation policies which can be attached to or detached from individual components during the execution. An adaptation policy is a set of rules modeled after the Event – Condition – Action rules. The development and execution of a component-based self-adaptive application in this framework can be decomposed in two phases:

1. Development of an adaptable application using an appropriate component model, without worrying about the details of the execution context it will be deployed in.

2. Definition of adaptation policies and binding of these policies to the components of the application by the deployer. This binding is dynamic, hence adaptation policies can be defined and attached to components at run-time.

This thesis focuses on the first phase of this development process, wherein the specification of such adaptive components is created. The thesis proposes that, while the adaptation logic can be considered as a separate concern, it should still be indicated in the service or component specification. This makes the clients aware of the adaptive functionality provided by the service.

2.2.2.2. The K-Component Framework
Dowling et al, introduce the K-Component Framework [DOWL01] which suggests the use of dynamic software architectures and architectural reflection in building adaptive systems. Architectural reflection allows a system to inspect and modify its configuration graph of components and connectors at run-time. The reconfiguration operations over the architecture are implemented as graph transformations, guaranteeing the safety and integrity of the architecture both during and after reconfiguration. A graph transformation supported by the framework is a rule-based manipulation of the configuration graph. Rules define how and when a graph is transformed. The interfaces and connectors represent the vertices and edges in the graph. They describe the part of the system that is
preserved during a graph transformation. The components and connector properties represent the labels of the vertices and edges in the graph respectively. These labels describe the part of the graph that is rewritten during a graph transformation. Thus, their model of dynamic reconfiguration is constrained to replacing the components in a system’s configuration graph and changing the connector strategies.

The framework also provides a separate adaptation contract description language for writing reflective programs. These programs are called adaptation contracts which allow programmers to specify how and when to reconfigure the software architecture at runtime. The adaptation contract contains a series of conditional rules for the transformation of the software architecture’s configuration graph.

Thus this framework provides the way of achieving adaptation in a software system composed of different components. This thesis complements the work done by the framework by providing ways to represent these adaptations in the service specification. The adaptation contracts in this framework indicate how and when adaptations are performed. On the other hand, MSAS proposed in this thesis, indicate what adaptations are performed.

2.2.2.3. **MONSOON: A Coevolutionary Multiobjective Adaptation Framework for Dynamic Wireless Sensor Networks**

This framework is developed by Boonma et al. [BOON08] for Wireless Sensor Network (WSN) applications which are required to adapt their operations to dynamic changes in the environment. They propose BiSNET/e, (Biologically-inspired architecture for Sensor NETworks, evolutionary edition) a runtime architecture. It consists of two software components: agents and middleware platforms. Each WSN application is designed as a decentralized group of agents. Agents collect sensor data and/or detect an event on platforms on individual nodes. Then, they carry sensor data to base stations, in turn, to a backend
MONSOON is a co-evolutionary adaptation framework for agents. Each agent possesses its own behavior policy, as a *gene*, which defines how to invoke its behaviors. MONSOON allows agents to evolve their behavior policies via genetic operations (mutation and crossover) across generations and simultaneously adapt the behavior policies to conflicting objectives in dynamic physical operational environments and network environments. MONSOON considers three objectives: success rate, latency, and power consumption. The evolution process in MONSOON frees application designers from anticipating all possible environment conditions and tuning their agent’s behavior policies to the conditions at design time. Thus MONSOON framework allows agents to perform unanticipated adaptations. On the other hand this thesis only considers anticipated adaptations which can be represented in the service specification. Thus MSAS of an agent will represent all the anticipated adaptations.

2.2.2.4. A Framework for Dynamic Service Adaptation in the Grid

Weissman et al. address the problem of dynamic service adaptation in the Grid, using this framework [WEISS 05]. The framework presents an architecture and prototype implementation for dynamic Grid services that extends OGSA (Open Grid Service Architecture) to better support dynamic Virtual Organizations (VOs). The framework addresses the problem of dynamic service hosting - where to host and re-host a service within the Grid. Framework has developed several new adaptive Grid service classes that are designed to better capture the dynamics of the Grid. Dynamic service deployment enables services to be added or upgraded without “taking down” a site for reconfiguration. The adaptive Grid service (AGS) is the fundamental abstraction for a Grid service that can adapt to changes in demand and resource availability. The AGS consists of three components: a frontend, deployer, and back-end. AGS front-end handles client requests and makes decisions about where the request should run. The AGS deployer decides which site(s) should host and deploy the service. Information
about when a service is deployed and running is maintained by the front-end. The back-end consists of an AGS factory that contains the actual code for the service and serves each request by creating an AGSI (adaptive Grid service instance).

This thesis complements the work done by this framework by providing a multi-level specification for AGS which indicates its adaptations. This framework on the other hand provides the ways of achieving these adaptations.

2.2.3. Adaptive SOA
The Service-Oriented Architecture (SOA) [ERL 05] has been identified as an effective approach to build systems by composing inter-organizational services. These services are developed using different technologies. They are configured and published by providers who are responsible for provisioning the services with sufficient resources.

As suggested by Erl [ERL 05], the application logic in the context of SOA can be split into two layers: the service interface layer and the application layer. Service Interface Layer is where loosely coupled services communicate via open protocols. The application layer is where service application logic is developed and deployed on different technology platforms. Such services need to adapt to change in environment and requirements. Following framework proposes creation of such adaptive services using an adaptive framework for SOA.

2.2.3.1. Cross-layer Self-adaptation of Service-oriented Architectures
Gjørven et al., [GJø08] suggest efficient adaptation to run-time contextual changes in SOA–based systems. They propose the adaptation framework “QUA” for SOA-based systems which performs cross-layer adaptations. Cross-layer adaptation consists of adaptation of a system consisting of several layers. The technologies and mechanisms of each layer are integrated and controlled by
the same adaptation framework. In the context of SOA, it means coherent adaptation across the service interface and application layers of a SOA system. This is done while preserving the loose coupling and autonomy of the services. The QUA adaptation framework consists of a planning framework, which is responsible for selecting service implementations and configurations, a platform framework, which encapsulates mechanisms for managing and adapting services, and a supporting Service Meta-Object Protocol. This framework provides a QUA middleware, which supports self-adaptive SOA applications. It integrates interface layer and application layer mechanisms to provide different degrees of adaptation. It supports adaptations mechanisms such as service selection and composition. This middleware provides technology independent adaptation reasoning and adaptation strategies. Summarizing, this framework provides a middleware for SOA based applications which can adapt by selecting and composing different services. On the other hand, this thesis concentrates on assisting the development of individual adaptive services by providing their formal Multi-level Specification. This specification will assist the selection and composition process used by this framework.

2.2.4. Adapter Design pattern
The Adapter design pattern is used to enable objects with different interfaces to communicate with each other. Adapters come in two flavors: object adapters and class adapters. Object Adapters: Object adapters use a compositional technique to adapt one interface to another. The adapter inherits the target interface that the client expects to see, while it holds an instance of the adaptee. When the client calls the request() method on its target object (the adapter), the request is translated into the corresponding specific request on the adaptee. Object adapters enable the client and the adaptee to be completely decoupled from each other. Only the adapter knows about both of them.
Class Adapters: Class adapters use multiple inheritance to achieve the communication. As in the object adapter, the class adapter inherits the interface of the client's target. However, it also inherits the interface of the adaptee as well. The request to the target is simply rerouted to the specific request that was inherited from the adaptee interface. Class adapters are simpler than object adapters in that they involve fewer classes and are useful if total decoupling of the client and adaptee is not needed. While Adapter design pattern enables implementation of interface adaptation, MSAS enables representing this adaptation. Thus an adaptive service implementing the Adaptor pattern can be formally represented using MSAS proposed by this thesis.

2.3. Component Based Adaptation Techniques
Component adaptation is a widely researched problem. In this section we provide an overview of the component adaptation methodology proposed by the prior researchers.

All the following techniques provide methodologies for component adaptation or composition. They specify "How" services can perform adaptation. On the contrary this thesis provides way to specify different adaptations for a service in a detailed specification. This concise specification can thus help in component composition and provide an overview of "What" adaptations are supported by the service.

2.3.1. A Formal Approach to Component Adaptation and Composition by David Hemer
In [HEM05], David Hemer defines component adaptation and composition strategies using parameterized library templates. Hemer proposes the use of templates from the CARE language [LIN97] to define adaptation strategies for
modifying and combining components. The language is functional in nature, with a program consisting of units such as types and fragments. Types model data structures; fragments are similar to the functions used in functional programming languages. The templates in CARE language can be parameterized over functional behavior, as well as over types. Hemer defines a variety of adaptation templates such as wrapper templates and architecture templates. Wrapper templates adapt a single program component. Architecture templates combine program components. The wrapper template proposed by Hemer, relies on single fragment. It performs modifications on this fragment to provide a new fragment. They have given two examples of wrapper templates: one which modifies input arguments and other which modifies pre-condition of a function. The Sequential architectures template allows problems to be solved by combining fragments sequentially. They have given two examples: one is a syntactic based adaptation strategy, and other is a semantic based adaptation strategy. Independent architectures allow problems to be solved by splitting the problem into sub-problems that can be solved independently, such that each sub-problem does not rely on input or output from the other sub-problems. The final class of adaptation architecture templates is the alternative architectures. For this class of the templates, the main fragment is implemented by choosing between two or more other fragments.

As this thesis concentrates on designing specification for adaptive component or a service, it supports constructs for representation of wrapper. This is similar to wrapper templates proposed by Hemer. On the other hand, this thesis does not provide ways to specify adaptations provided by the architecture templates as these adaptations are applied over multiple services.

2.3.2. A formal approach to component adaptation

In [BRAC05], Braccaiali et al. present a formal methodology for adapting components with mismatching interaction behavior. The three main elements of the methodology are: (1) the inclusion of behavior specifications in component
interfaces, (2) a simple, high-level notation for expressing adaptor specifications, and (3) a fully automated procedure to derive concrete adaptors from given high-level specifications.

The approach focuses on the problem of adapting mismatching behaviors of components, considering the aspects such as the component interface, the adaptor specification and the adaptor derivation. The component interface is based on IDL. IDL interfaces are extended with a description of the behavior of the components. Thus, an interface consists of two parts: A signature definition (describing the functionalities offered and required by a component), and a behavior specification (describing the interaction protocol followed by a component). Signatures are expressed in the style of traditional IDLs and behavior specifications are expressed by using a subset of π-calculus. The paper presents a notation for expressing the specification of an adaptor which represents the interoperation of two components. The adaptor specification consists of a set of correspondences between actions and parameters of the two components. This notation produces a high-level, partial specification of the adaptor. A concrete adaptor is automatically generated, given its partial specification and the interfaces of two components. This is done by exhaustively trying to build a component which satisfies the given specification. The separation of adaptor specification and derivation allows for automating the error-prone, time-consuming task of generating a detailed implementation of a correct adaptor, thus simplifying the task of the software developer.

Braccialli et al.’s approach deals with composing two components, given an adaptive specification. Thus, it deals with adaptation of an already existing component for integration with other components. On the other hand this thesis proposes considering adaptation right from the design of a component or service. The adaptation specification proposed by Braccialli et al. is different from a MSAS. Braccialli et al.’s adaptive specification consists of methodology for specifying the required adaptation between two components. It provides a
mapping among the functionalities of two components to be adapted. On the contrary, MSAS proposed by this thesis, provide a formal specification for an individual adaptive service representing the adaptations the service can perform.

2.3.3. Model-Based Adaptation of Behavioral Mismatching Components

Canal et al, [CAN08] propose the generation of adaptors to compensate for a mismatch between component interfaces. The approach is based on an abstract notation based on synchronous vectors and transition systems for governing adaptation rules. It introduces a model-based adaptation approach focusing on mismatch appearing at the behavioral level. It also addresses name mismatch at the signature level. The approach takes as input the behavioral interfaces of components to be adapted and an adaptation contract which is an abstract description of the constraints which must be respected to make the involved components work together. Given these two elements, an adaptor protocol is generated in an automatic way. They have used synchronous vectors as adaptation contract language to explicitly specify the interactions between components, possibly on different message names. Their notation also allows the specification of ordering constraints on interactions between components. In order to generate adaptor protocols for such contracts, they have introduced two algorithms that automate the adaptation process.

Thus, Canal et al’s approach deals with automatically generating adaptors between software components. On the other hand work in this thesis assists designing individual adaptive components or services. The thesis designs constructs for representing the behavioral and signature level adaptations performed by an individual adaptive service. The Adaptation contract introduced by canal et al. describes interactions of different components and their associated constraints. This is different than an adaptive interface for a service proposed by this thesis, as it describes the adaptations performed by a single service.
2.3.4. Automating Component Adaptation for Reuse

Morel et al. [MOREL03] present a framework, called SPARTACAS, for automating specification-based component retrieval and adaptation. In this framework, components that partially satisfy the requirements of a design problem are adapted using adaptation architectures. Adaptation architectures modify the behavior of a software component by enforcing interactions with other components. SPARTACAS is used to retrieve possible solutions to a problem from a library of components. The framework uses formal specification based on axiomatic structure. It uses a feature-based, signature-based and specification-based retrieval engines for retrieving matching components. These components can be completely or partially matching. It then uses a component adaptation framework which contains a collection of adaptation architecture theories for adapting behaviors of components. The behavioral adaptation proposed by SPARTACAS involves altering the functionality of a component through its interactions with other components. SPARTACAS supports three different adaptation architectures: sequential, alternative, and independent.

Again this framework deals with adapting behaviors of already existing components. On the other hand this thesis proposes adaptation to be considered right from design of the service. If components are designed for adaptation, then the process of retrieval of matching components from the library is accelerated. As this retrieval is based on a component specification which indicates its adaptive behavior. Thus the adaptation required for making components interact will be done by the adaptive component itself.

2.3.5. Superimposition: A Component Adaptation Technique

Bosch proposes, Superimposition [BOS99], a technique for the black box adaptation that imposes predefined, but configurable types of functionality on reusable components. In [BOS99], Bosch has identified and classified different component adaptation techniques such as copy-paste, inheritance and wrapping.
The paper has also identified requirements for component adaptation techniques such as transparency, black-box, composable, configurable, and reusable. Paper defines object superimposition $S$ of $B$ over $O$ as the additional overriding behavior $B$ over the behavior of component $O$. It has identified a set of reusable component adaptation types which can be superimposed over existing components. Paper presents three different categories of component adaptation, component interface changes, component composition, and component monitoring. Typical examples proposed by Bosch for component interface adaptation are, changing operation names, restricting parts of the interface, clients and state-based restriction. For each of these types he has defined superimposing entities to satisfy the respective adaptation. Similarly, he has identified superimposing entity types for delegation of requests, component composition, acquaintance selection and binding. Bosch proposes the component monitoring category of adaptation for monitoring of the component so that other components are notified or invoked when certain events at the monitored component occur. He discusses three examples of monitoring that can be superimposed on reusable components, implicit invocation, observer notification and state monitoring.

Thus using Bosch’s theory, different adaptation types can be created and superimposed on already existing reusable components transparently. Hence, Superimposition is a way of achieving component adaptation. On the other hand, the work done by this thesis enables creation of adaptive services and their interfaces. Thus adaptations specified in MSAS of an adaptive service can be implemented using techniques such as Superimposition.

2.3.6. Module Reuse by Interface Adaptation

Nimble language [PURT91] proposed by Purtilo et al., provides a new pattern-based language for rewriting the arguments and return values of procedure call. It allows programmers to adapt the interfaces of existing software without manual
manipulation of the source. This allows reuse of broader range of software components. It can reorder, duplicate or omit arguments, or supplement them based on default values, and can also call on external functions where necessary to convert representations and types. The Nimble language is a declarative language allows programmers to provide the Nimble map, which is based on the pattern of parameters in the formal and actual interfaces. Along with the design of these maps, Nimble has provided translators for generating the interface codes based on those maps.

Thus, Nimble allows reuse of existing components by providing the necessary adaptations. On the other hand, the work done by this thesis defines different adaptations performed by an adaptive service at every level of service specification. Work done in the Nimble language for adapting the interfaces is very close to the adaptations proposed by this thesis at the Syntax level of specification. The difference is that, the Nimble language performs these adaptations after the component is developed, and this thesis suggests consideration of these adaptation right from design of the service. In addition to the syntax level interface adaptation, this thesis proposes adaptation at semantic, synchronization and QoS level of specification.

Conclusively, this chapter focuses on identifying and distinguishing the work done by researchers in the field of adaptation of software components and the work done by this thesis. The merit of the approach taken by this thesis is that, it suggests design for change. The thesis proposes development of adaptive software services and their formal specification. On the other hand, the demerit of this approach compared to this research work is that, it does not consider adaptation of already existing components.
CHAPTER 3. DESIGN AND IMPLEMENTATION

This chapter discusses the design of MSAS, which involves addition of new constructs to the multi-level service specification to represent an adaptive service. The chapter starts with defining an adaptive service in section 3.1. Section 3.2 discusses the UMM specification [RAJ00] on which MSAS is based. Section 3.3 then identifies the adaptive constructs for a multi-level specification and illustrates using distributed information classifier service interface starting from syntax to semantics, synchronization and QoS levels.

3.1. Definition of Adaptive Service

Stephen Kell [KEL08] defines software system adaptation as “any process which modifies or extends the implementation or behavior of a subsystem to enable or improve its interactions, or synonymously, its communication with the surrounding parts of the system (which we call its environment). Note that “communication” here includes not only dynamic interactions at run-time, but also interactions occurring statically, perhaps in the compiler”.

Thus, a service can perform different adaptations during its lifecycle. Of all these adaptations provided by the service, clients are interested only in a subset. Only those adaptations that directly or indirectly affect the contract of the service are part of this subset. Hence, while designing the specification for an adaptive service only this subset of adaptations should be considered. As indicated earlier, the contract of a service, as defined by Beugnard[BEU99] et al., consists of multiple levels such as syntax, semantics, synchronization and QoS. Therefore, this thesis considers adaptations of a service affecting these levels of
the contract only. It considers the design of an adaptive specification of a service in a level by level manner.

Agents in the domain of Artificial Intelligence incorporate special type of adaptations implemented using learning techniques. A learning technique is a way of modifying or extending the behavior of a service. In this research, modifying the behavior of a service using the learning techniques is also considered as a type of adaptation. As these learning techniques affect the contract of the service and hence, should be indicated in a service’s specification. Thus, this research defines an adaptive service to be, “a service which modifies or extends its implementation or behavior in such a way that it affects its contract with the outside world”.

Based on this definition, following types of adaptations are considered in the design of a multi-level adaptive specification of a service:

- Syntax Adaptation
- Semantic Adaptation
- Synchronization Adaptation
- QoS Adaptation
- Adaptation by learning.

3.2. UMM Specification

In [RAJ00], Raje et al. suggest an enhancement of the multi-level contracts of Beugnard [BEU99] called as UMM specification of components. Based on UMM [RAJ00], a multi-level specification of a service can be represented using a feature diagram [LEE02] as in figures [3.1, 3.2, 3.3]. As indicated in the diagrams, service’s multi-level specification consists of mandatory features such as Inherent Attributes, Functional Attributes, Functions &Contracts. The Functions &Contract component specifies syntax, semantics, synchronization and QoS contracts for all the functions provided by the service [RAJ00].
Figure 3.1: Feature diagram for multi-level service specification
Figure 3.2: Feature diagram for Inherent Attributes of a service

Figure 3.3: Feature diagram for Functional Attributes of a service
As indicated earlier, there is a need to modify or extend this multi-level specification with new constructs to describe an adaptive service. Following sections describes the design.

3.3. Multi-level Specification of an Adaptive Service
The multi-level specification of an adaptive service is created by enhancing the UMM specification of a non-adaptive service. In order to describe these enhancements and their associated rationale, a running example of an information classifier service is used throughout the remainder of this chapter.

3.3.1. Information Classifier Service
An information classifier service (referred as a “classifier”) is a typical service used in the domain of distributed information filtering [RAJ97]. This service provides document classification with the help of a predefined thesaurus containing keywords for a specific domain. It also communicates with a set of other classifiers, called as its acquaintances, on the network. If a classifier is
unable to classify a document by itself then it sends the document for classification to its acquaintances.
The classifier provides following function to the clients.

Table 3.1: Signature of classify function of the classifier service

```plaintext
DocumentClass classify (Document doc)  
/* This method classifies a given document and returns the result of the classification in form of a class to which that document belongs.*/
```

The UMM specification of such a classifier is shown in Table 3.2.

Table 3.2: UMM specification of the classifier service.

<table>
<thead>
<tr>
<th>Name</th>
<th>Classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>ComputationalAttributes,</td>
</tr>
<tr>
<td></td>
<td>CooperationAttributes, AuxiliaryAttributes.</td>
</tr>
<tr>
<td></td>
<td>ComputationalAttributes: InherentAttributes,</td>
</tr>
<tr>
<td></td>
<td>FunctionalAttributes.</td>
</tr>
<tr>
<td>InherentAttributes</td>
<td>Author, Version, DateDeployed,</td>
</tr>
<tr>
<td></td>
<td>ExecutionEnvironment, ComponentModel, Validity, Structure, Registrations.</td>
</tr>
<tr>
<td>Author</td>
<td>Sucheta Phatak.</td>
</tr>
<tr>
<td>Version</td>
<td>1.0.</td>
</tr>
<tr>
<td>DateDeployed</td>
<td>01-21-1983.</td>
</tr>
<tr>
<td>ExecutionEnvironment</td>
<td>N/A.</td>
</tr>
<tr>
<td>ComponentModel</td>
<td>N/A.</td>
</tr>
<tr>
<td>Validity</td>
<td>N/A.</td>
</tr>
</tbody>
</table>
Structure:: N/A.
Registration:: N/A.
FunctionalAttributes:: TaskDescription, AlgorithmmandComplexity, Architecture, Alternatives, Resources, Designpatterns, Usages, Aliases, FunctionsAndContract.
TaskDescription:: “Classifies given document”. AlgorithmmandComplexity:: O(n)
Architecture:: Multiprocessor, Uniprocessor.
Speed:: 4.
Load:: 10.
Alternatives:: N/A.
Resource:: Architecture, Load, Speed.
Designpatterns:: N/A.
Aliases: Search Agent; Document Classifier
FunctionAndContracts:: Function, BehavioralContract, ConcurancyContract.
Function:: DocumentClass Classify (Document doc) throws FileNotFoundException, RemoteException.
BehavioralContract:: Precondition, PostCondition, Invariant
Precondition:: doc != null.
PostCondition:: returnDocumentClass !=null.
Invariant:: Acquaintance>0
ConcurancyContract:: Single-Threaded.
CooperationalAttributes:: N/A.
AuxiliaryAttribute:: QualityOfService.
QualityOfService:: TurnAroundTime, Availability, Accuracy.
TurnAroundTime:: 45.
Availability:: 100%
Accuracy:: 50%
This thesis considers the XML version of UMM Specification and extends it to represent an adaptive service.

3.3.2. Design of Specification for Adaptive Service

The classifier service given in previous section is a plain service which does not provide any adaptations. On the other hand, if the requirements and the execution environment demand the service to adapt, then this plain service will not be able to cater to it. Hence this thesis suggests that, if adaptation is required from the service, then it should be considered right from the design. The service developer should first start by creating an adaptive specification for the service, there by designing the service for change.

This section describes the design of multi-level specification of an adaptive service by considering adaptation at all the levels of specification. At each level of specification (syntax, semantics, synchronization and QoS), possible adaptations and constructs to indicate these adaptations are identified. Subsequently, the impact of special type of adaptation using learning techniques on the adaptive specification is also discussed. For each identified adaptation at level \( i \), a relation (transformation) \( R_i \) between the basic (default) feature and adaptive feature is identified. The relation \( R_i \) from feature \( a \) to \( b \) is written using the notation suggested by [ROS6], as \( a R_i b \). This indicates \( a \) is related to \( b \) by \( R_i \).

This relation can be mapped to a function \( \delta_i \), which defines the transformation from \( a \) to \( b \). \( \delta_i \) is denoted as \( \delta_i(a) = b \), which means \( \delta_i \) transforms the feature \( a \) to \( b \). For each pair of \( a \) and \( b \) belonging to \( R_i \), the service has to implement \( \delta_i \). This is because relations are generalization of functions [ROS6]. A function represents a relation where exactly one element of the domain is related to each element of the source. Once \( \delta_i \) is known, cost or penalty associated with adaptation can be deduced.
3.3.2.1. Syntax Adaptation
Syntax is the basic level of a service contract, which specifies how clients can communicate with the service. This level indicates, for each function or operation of the service, the function name, input and output parameters with their types, and possible exception that may be raised during a function call.
This is represented in the syntax level XML specification as:

Table 3.3: Syntax specification of a function in XML format

<table>
<thead>
<tr>
<th>Function Name=“Function”</th>
<th>Return Type=“Type of return variable” Name=“Name of return variable”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>Argument Type=“Type of argument” Name=“Name for the argument”</td>
</tr>
<tr>
<td></td>
<td>....</td>
</tr>
<tr>
<td>&lt;/Arguments&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Hence, the Syntax level XML specification of the classifier service will look like:

Table 3.4: Syntax specification of the classify function in XML format

<table>
<thead>
<tr>
<th>Function Name=“Classify”</th>
<th>Return Type=“DocumentClass” Name=“result”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>Argument Type=“Document” Name=“doc”</td>
</tr>
<tr>
<td>&lt;/Arguments&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/Exceptions&gt;</td>
<td>FileNotFoundException; RemoteException</td>
</tr>
</tbody>
</table>

</Exceptions>
A service can adapt the syntax of its functions depending of its requirements. This thesis identifies following adaptations of a service at the syntax level.

- Name Adaptation
- Type adaptation
- Adaptation in number and order of arguments

The exceptions list of an adaptive function should include any exceptions thrown by the function while handling these adaptations.

3.3.2.1.1. Name Adaptation

Functions provided by a service can adapt their name. This adaptation enables supporting clients which looking for a specific name of given service function. Given name of a function can be adapted to its synonym. All such supported synonyms can be listed in the adaptive interface or can be referred from a dictionary that stores synonyms as the values and name as the keyword.

Thus, an original function name, $F_O$, can be adapted to, $F_A$, where the relation, $R_F(F_O R_F F_A)$, is such that $F_A$ is a synonym of $F_O$. Corresponding function $\delta_F$ to be implemented by the service to achieve this relation is implementation of the function with name $F_A$.

This name adaptation can be represented in the service specification at the syntax level. For each function, $f$, in the functions & contract component of the service, if $f$ supports name adaptation, then its syntax will be represented using following notation:
Thus, if the classifier service adapts name of the function classify to "Sort", "Assort", "getClassification" its syntax level adaptive specification is presented in Table 3.6.

Table 3.6: Syntax specification of the classify function supporting adaptive names.

```
<Function Name="Classify">
   <Adapts>Sort;Assort;getClassification</Adapts>
   <Return Type="DocumentClass" Name="result" />
   <Arguments>
      <Argument Type="Document" Name="doc"/>
   </Arguments>
   <Exceptions> FileNotFoundException; RemoteException </Exceptions>
</Function>
```

3.3.2.1.2. Type Adaptation

Functions of a service can also adapt by accepting different argument types. A function can also adapt by changing its return type according to clients requirements. Thus, the adaptation in return or arguments types is termed as Type adaptation. This adaptation affects the syntax of the service and hence, appropriate constructs should be provided at the syntax level of the specification to indicate this change.
Let the argument or return type accepted by a function, $f$, of a service be denoted by $T_O$. This type can be adapted to a new type $T_A$. This adaptation is achieved by the use of a function $R_T$. Following forms of relation $R_T$ are identified. $T_O \ R_T \ T_A$, where $R_T$ can be

- Super class relationship.
- Conversion relationship.

If $R_T = \text{Superclass relation}$, i.e., $T_A$ is a super class of $T_O$ ($T_O$ forms the base class of $T_A$). The corresponding $\delta_T$ required to transform $T_O$ to $T_A$ to satisfy this relation is None. This is because any instance of the derived class is always an instance of the base class. Hence, any function accepting a base class object will accept a derived class object without any conversion.

If $R_T = \text{Conversion relation}$, i.e., $T_A$ can be transformed or casted or converted (either by coercion or an explicit casting) to $T_O$, then $\delta_T$, is the function which accepts an object of $T_O$ and returns the corresponding $T_A$ object.

Type adaptation for arguments or return type is represented using following notation. For each function $f$ of the adaptive service supporting Type adaptation, the syntax level will be specified as follows,
Table 3.7: Syntax specification of a function supporting Type Adaptation

<Function Name="fun_name">
  <Return Type="##Type of return variable" Name="##Name of return variable">
    <Adapts> ##Comma separated list of adapted types</Adapts>
  </Return>
  <Arguments>
    <Argument Type="##Type of argument" Name="##Name for the argument">
      <Adapts> ##Comma separated list of adapted types
    </Adapts>
    </Argument>
    #More arguments
  <Arguments>
</Function>

Thus, the adaptive classifier supporting type adaptation for the classify function can be represented at syntax level as:
As depicted in the above specification, this adaptive function classify, adapts its return type to String, here the relation between "DocumentClass" and String is "Conversion" relation. Similarly, the argument doc adapts to type “URL” instead of the type “Document”.

3.3.2.1.3. Adaptation in number and order of arguments
A function can adapt in the number of arguments it accepts and also the order of these arguments. By supporting this adaptation, clients can make a call to this adaptive function with an arbitrary order of input arguments and/or dropping one or more arguments. To simplify this type of adaptation, this thesis assumes without loss of generality that, clients call the function using named parameters or keyword arguments. Many languages such as Ada, Common Lisp, Modula-3, Fortran 90 support named parameters [SEB07]. Function calls using named parameters or keyword arguments clearly state the name of each argument in the function call itself.
Let the original keyword argument list accepted by the function \( f \) be \( A_O \) and the adapted keyword argument list be \( A_{OA} \), then the adaptation in the order of keyword arguments is specified by a relation \( R_O \) between \( A_O \) and \( A_{OA} \):

\[
A_O \; R_O \; A_{OA}, \text{ where } |A_{OA}| = |A_O| = n, \text{ where } |A| \text{ indicates cardinality of } A, \text{ and },
\]
\( A_{OA} \) is an \( n \)-permutation [ROS6] of \( A_O \).

Let \( \delta_O \) is the function implemented by the service to achieve relation \( R_O \). This \( \delta_O \) depends on the language in which the service is implemented. If this language supports named parameters then \( \delta_O \) will be “None”. On the other hand, if the language does not support named parameters, then \( \delta_O \) represents the transformation needed to translate adaptive argument order in the required order.

Along with adapting in the accepted order of arguments, a function can adapt in the accepted number of arguments. This means clients can choose to not pass some or all arguments in the argument list. These omitted parameters, in such cases, will assume default values. If some arguments of a function are mandatory parameters then those have to be specified by the clients in the function call. Thus, an adaptive function can only adapt number of default arguments accepted. Assumption here is that a given argument list of an adaptive function is exhaustive and hence, that function will not consider adapting the number of arguments by adding new arguments. Let the original argument list be \( A_O \) and the adapted argument list \( A_N \), then the adaptation in number of arguments is specified by a relation \( R_N \):

\[
|A_O| = n, \; |A_N| = m, \; n \geq m, \text{ where } |A| \text{ indicates cardinality of } A, \text{ and },
\]
\( A_O = A_C \cup A_D, \; A_C \) is the set of all core arguments in \( A_O \) and \( A_D \) is the set of all default arguments in \( A_O \),
\( A_O \; R_N \; A_N, \) is such that,
\( A_N = A_C \cup A_{DN}, \; A_{DN} \) is a combination [ROS6] of \( A_D \).
Corresponding $\delta_N$ is a function implemented by the service to achieve $R_N$. This $\delta_N$ depends on the language in which the service is implemented. If this language supports default arguments then $\delta_N$ will be "None". On the other hand, if the language does not support default arguments, then $\delta_N$ represents the transformation needed to provide default arguments. A function supporting order and number adaptation for its arguments is represented as:

| Function Name="##name of the Function">  
| Return Type="##Type of return variable" Name="##Name of return variable"> 
| <Adapts> ##Comma separated list of adapted types</Adapts> 
| </Return> 
| <Arguments> 
| <Argument Type="##Type of argument" Name="##Name for the argument" Default ="true"> 
| <Adapts> ##Comma separated list of adapted types 
| </Adapts> 
| </Argument> 
| ##More arguments 
| <Adapts order="true"/> 
| <Arguments> 
| </Arguments> 
| </Function>

Thus, for arguments which are not mandatory the default attribute is set to true. For indicating order adaptation of arguments, "Adapts" element with attribute "order" is added to the "Arguments" element.
Thus the adaptive classifier service, which accepts a thesaurus along with the document to be classified, can be represented as:

Table 3.10: Syntax specification of the classify function supporting order and number adaptation

```
<Function Name="Classify">
  <Adapts>Sort, Assort, getClassification</Adapts>
  <Return Type="DocumentClass" Name="result" >
    <Adapts> String </Adapts>
  </Return>
  <Arguments>
    <Argument Type="Document" Name="doc">
      <Adapts> URL </Adapts>
    </Argument>
    <Argument Type="Thesaurus" Name="thesaurus" Default="NetworkThesarus.txt" />
    <Argument Type="int" Name="return_Type" Default="0"/>
    <Argument Type="int" Name="postProc" Default="0"/>
    <Adapts Order="true"/>
  </Arguments>
  <Exceptions> FileNotFoundException; RemoteException </Exceptions>
</Function>
```

As specified above, the thesaurus argument is default and if the client does not provide it, a default thesaurus is used. The arguments return_Type and postProc used in this specification of classify function are defined in Section 4.2.2.1 and 4.3.2 respectively. The order of these arguments is adaptive and hence, this classifier service accepts any order of these four arguments.
3.3.2.2. Semantic Adaptation

The multi-level interface of a service includes the semantics of each function provided by the service. These semantics are represented using logical assertions for:

- Pre-condition
- Post-condition
- Invariant

These assertions describe the behavior of the function. Pre-condition describes what the function requires before a call to this function is made. Post-Condition describes what the function ensures after the function is executed. Invariant describes what the function does not change during its execution. Thus for each function provided by a service, its semantic level of the XML specification is represented as:

Table 3.11: Semantic specification of a function

```xml
<Semantics>
  <PreCondition Expression="##Logical Assertion for the pre-condition" />
  <PostCondition Expression="##Logical Assertion for the post-condition" />
  <Invariant Expression="##Logical Assertion for the invariant" />
</Semantics>
```

Hence, the semantic level of the specification of classifier service using XML syntax is:
As indicated earlier, the adaptation is a modification or an extension of the behavior of a service. Although, an adaptation is a modification, this modification is within the context of the service, i.e., it does not change the core functionality of the service but provides a logical extension to it. Hence the semantic adaptation is identified as a wrapper adaptation [BOS99] which involves pre-processing and/or post-processing over the core functionality of the service. Following section identifies the wrapper adaptations and their representation in adaptive interface.

3.3.2.2.1. Wrapper Adaptation

A wrapper adaptation [BOS99] for a service is done as a pre-processing activity or a post-processing activity or a combination of both. Preprocessing can be done before a function executes. This can represented using an adaptive pre-condition for the function as described below.

Adaptive Pre-Condition

Let $S_{pre}$ be the pre-condition of a given function in a service. A service can provide a wrapper adaptation using pre-processing wherein it accepts a weaker pre-condition than $S_{pre}$ at some cost. Dijkstra [DIJK76] suggests that a pre-condition for a function should be the weakest pre-condition ($S_{pre}$) that the function accepts. Any stronger pre-condition than this weakest pre-condition is by
default accepted by the function. A stronger pre-condition than $S_{Pre}$ will not need a change in the service functionality. On the other hand, if a function wants to support a weaker pre-condition than $S_{Pre}$ then a change or an extension of the current functionality is needed. This also means performing this behavior change would impact performance (as depicted by QoS parameters) of the function. Such a weaker pre-condition is the adaptive pre-condition for that function. Let $A_{Pre}$ be an adaptive pre-condition accepted by the function then the relation $R_{Pre}$ between $A_{Pre}$ and $S_{Pre}$ is such that:

$$S_{Pre} R_{Pre} A_{Pre}, \text{ where } S_{Pre} \rightarrow A_{Pre}, \text{ i.e. } S_{Pre} \text{ logically implies [ROS6] } A_{Pre}.$$ 

The corresponding $\delta_{Pre}$ for $R_{Pre}$, represents a state transformation of the service requiring $A_{Pre}$ and ensuring $S_{Pre}$. In other words, the pre-condition of $\delta_{Pre}$ is $S_{Pre}$ and the post-condition is $A_{Pre}$.

The semantic level of the contract for a function of a service supporting adaptive pre-condition is:

| Table 3.13: Semantic specification of a function supporting adaptive pre-condition |
|<Semantics>|<PreCondition Expression="##Logical Assertion for the pre-condition" >|<Adapts> ##Comma separated list of adaptive pre-conditions </Adapts> </PreCondition>|<PostCondition Expression="##Logical Assertion for the post-condition" />|<Invariant Expression="##Logical Assertion for the invariant" />|</Semantics>
Hence, the semantic level of the adaptive specification of the classifier service is:

Table 3.14: Semantic specification of the classify function which supports adaptive pre-condition

|Semantics>|<PreCondition Expression="doc!=null && doc.getClass().toString().equals("Document") && doc.getType().equals("Text")"> <Adapts> doc !=Null && doc.getClass().toString().equals("Document") && (doc.getType().equals("Text") || doc.getType().equals("MS Word Doc")) </Adapts> </PreCondition> <PostCondition Expression="result != Null" /> <Invariant Expression="Acquaintance != Null" />

Thus the adaptive classifier accepts a URL for a document and downloads the document before executing the core functionality. Similarly, wrapper adaptation for a service can be done as a post-processing activity. Post-processing can be done after a function executes. This can be represented using an adaptive post-condition for the function as described in the following section.

Adaptive Post-Condition

Let $S_{Post}$ be the post-condition of a function of the service. At the end of function execution $S_{Post}$ is ensured. As suggest by Dijkstra [DIJK76], the post-condition of a function should be the strongest post-condition it can ensure. Now, if a client
wants the function to ensure a weaker post-condition that $S_{Post}$, then it will be by
default ensured without any functionality change. On the other hand, if a client
requires the function to ensure a stronger post-condition that $S_{Post}$, then the
function needs to change its behavior. The function can adapt to such post-
conditions by providing post-processing after its execution. Let $A_{Post}$ be such a
stronger post-condition, which is an adaptive post-condition. The relation, $R_{Post}$
between $S_{Post}$ and $A_{Post}$ can be represented as:

$$S_{Post} R_{Post} A_{Post}, \text{ where } A_{Post} \rightarrow S_{Post}, \text{ i.e. } A_{Post} \logically \ implies \ [ROS6] \ S_{Post}.$$

The corresponding $\delta_{Post}$ associated with this $R_{Post}$ represents a state
transformation requiring $S_{Post}$ and ensuring $A_{Post}$. In other words, pre-condition of
$\delta_{Pre}$ is $S_{Post}$ and post-condition is $A_{Post}$. Hence, the semantic level of the contract
for a function of a service supporting adaptive post-condition is:

Table 3.15: Semantic specification of a function supporting adaptive post-
condition

```
<Semantics>
  <PreCondition Expression="##Logical Assertion for the
  pre-condition" />
  <PostCondition Expression="##Logical Assertion for the
  post-condition" >
    <Adapts> ##Comma separated list of adaptive
    post-conditions </Adapts>
  <PostCondition/>
  <Invariant Expression="##Logical Assertion for the
  invariant" />
</Semantics>
```

Hence, the semantic level of adaptive specification of the classifier service is:
Table 3.16: Semantic specification of the classify function which supports adaptive post-condition

<table>
<thead>
<tr>
<th>Semantics</th>
</tr>
</thead>
</table>
| <PreCondition Expression="doc!=null &&
  doc.getClass().toString().equals("Document") &&
  doc.getType().equals("Text")">
| <PostCondition Expression="result != Null">
| <Adapts> result!= null &&
  !result.getDocumentClass.IsEmpty()
| </Adapts>
| </PostCondition>
| <Invariant Expression="Acquaintance != Null" />
| </Semantics>

Thus, the adaptive classifier can adapt to a stronger post-condition where it not only ensures that result is not null but also ensure that result is always classified. This can be done by putting all unclassified documents under the “default” class.

Thus, semantic level adaptation is identified as adaptive pre-conditions and adaptive post-conditions. MSAS does not consider invariants to be adaptive, because adaptive invariants will cause change in core functionality of the service. This change should be reflected with a better design than an adaptive invariant.

3.3.2.3 Synchronization Adaptation
The synchronization level specification of a service represents synchronization related parameters or features of the service. It indicates type of synchronization, synchronization policies, synchronization techniques used by the service. As
suggested by Anjali Kumari[ANJ04], following information is provided by the synchronization level specification:

1. Synchronization policy Name – this feature indicates the name of synchronization policy used by the service’s interface to handle multiple client accesses/requests on a method. This policy can be one of the already defined policies in a Synchronization policy catalog [ANJ04] or it can be a user defined policy which extends one of the policies in the catalog.

2. Synchronization policy’s implementation technique – This feature indicates the technique used for implementing this policy. All possible implementation techniques are specified in the catalog and hence, this parameter can take one of the values from the catalog. For instance, the policy mutual exclusion can be implemented using semaphores or mutexes.

3. The synchronization pre-condition of the function – This parameter is required if the given policy is not defined in the catalog, but is extended from already existing policy. The synchronization pre-condition of a function specifies the synchronization pre-condition that the caller and the other methods of the server component must satisfy in order to start the execution of this function.

4. The synchronization action for the function – This parameter is also optional and is required only in case of extended policies. The synchronization action specifies the action with respect to synchronization that will be taken by the service’s function if the pre-condition is satisfied.

5. The synchronization invariants for the function – This parameter is required only in case of extended policy. The synchronization invariants specify the conditions that must hold throughout execution of the function.

6. The post-conditions of the methods — This parameter is required only in case of extended policy. The synchronization post-condition of a function specifies the conditions that must hold after the function is terminated.

Thus, the synchronization level of specification of a service is:
Table 3.17: Synchronization specification of a function

Here, the Name of the policy, can be of the form "<Name>:<BasePolicy>" where Name is the name of policy which extended is from a base policy presented in the policy catalog. Also, in case if a policy is one of the cataloged policies then, BasePolicy, Pre-Condition, Post-Condition, Action, and Invariant parameters are optional and can be referred from the catalog.

Using this specification format, if the classifier service function, “classify”, implements the mutual exclusion policy, then its synchronization level of specification will be:

Table 3.18: Synchronization specification of the classify function
An adaptive service can adapt to different synchronization policies or techniques according to a user request or the environment change. Hence, an adaptive service specification should indicate all such possible adaptations. This section identifies and describes two types of adaptation at this level of specification. First is the adaptation in the synchronization policy and second is the adaptation of the synchronization technique.

3.3.2.2.2. Synchronization Policy Adaptation

Let the given synchronization policy for a service’s function be $P_O$ and an adaptive synchronization policy be $P_A$. Let $Pre_O$, $Post_O$, $Inv_O$ be the pre-condition, post-condition and invariant of $P_O$ respectively. Similarly, let $Pre_A$, $Post_A$, $Inv_A$ be the pre-condition, post-condition and invariant of $P_A$ respectively. Then the following relation can be identified between $P_O$ and $P_A$.

$$P_O R_P P_A$$

Where $R_P$ is one of the following

- **Equivalence relation:**
  
  iff
  
  $Pre_O \Leftrightarrow Pre_A$,
  $Post_O \Leftrightarrow Post_A$
  $Inv_O \Leftrightarrow Inv_A$

- **Constrained:** i.e., $P_O$ is a more constrained policy than $P_A$
  
  iff
  
  $Pre_O \Rightarrow Pre_A$,
  $Post_O \Rightarrow Post_A$
  $Inv_O \Leftrightarrow Inv_A$

- **Relaxed:** i.e., $P_O$ is a more relaxed policy that $P_A$
  
  iff
  
  $Pre_O \leq Pre_A$,
  $Post_O \leq Post_A$
\[ \text{Inv}_O \Leftrightarrow \text{Inv}_A \]

If \( R_P \) is equivalence relation then, \( P_O \) is equivalent to \( P_A \), and hence, while adapting to \( P_A \) from \( P_O \), no change in synchronization behavior is required. Thus, \( \delta_{\text{Sync}} \), which indicates the transformation in synchronization behavior, is “Empty”.

On the other hand, if \( R_P \) is either Constrained or Relaxed relation then to adapt from \( P_O \) to \( P_A \), the function needs to change its synchronization behavior. This change in turn means the \( \delta_{\text{Sync}} \) associated with \( R_P \) has to implement this transformation.

Thus, the synchronization level specification of an adaptive service function is:

Table 3.19: Synchronization specification of a function supporting synchronization policy adaptation

<table>
<thead>
<tr>
<th>Synchronization type=&quot;##Type of Synchronization&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Policy Name=&quot;##Name of the Policy&quot;&gt;</td>
</tr>
</tbody>
</table>
|       <Adapts>"##Name of adaptive policy</Adapts>
| </Policy>                                      |
| <Implementation Technique="##Name of the    |
| implementation technique"/>                   |
| <PreCondition>"##Logical Assertion for the     |
| pre-condition </PreCondition>                 |
| <PostCondition>"##Logical Assertion for the    |
| post-condition</PostCondition>                |
| <Action> "##Action taken by the service </Action>|
| <Invariant> "##Logical Assertion for the      |
| invariant</Invariant>                         |
| </Synchronization>                           |

Let the classifier service function “classify”, which implements mutual exclusion policy needs to adapt to “FCFS” policy. Following table indicated the relation between mutual exclusion and FCFS policy.
Table 3.20: Relation between Mutual Exclusion policy and FCFS policy

<table>
<thead>
<tr>
<th>Policy Name</th>
<th>Mutual Exclusion</th>
<th>FCFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Condition</td>
<td>exec.func == 0</td>
<td>exec.func == 0 &amp;&amp; wait.func.time &gt;= this.time</td>
</tr>
<tr>
<td>Post-Condition</td>
<td>exec.func == 0</td>
<td>exec.func == 0</td>
</tr>
<tr>
<td>Invariant</td>
<td>exec.func &lt;= 1</td>
<td>exec.func &lt;= 1</td>
</tr>
</tbody>
</table>

Thus, the relation between Mutual Exclusion and FCFS is Relaxed. To support this adaptation, classify function needs to provide change in its synchronization behavior. The synchronization level specification for such an adaptive classify function is:

Table 3.21: Synchronization specification of the classify function supporting synchronization policy adaptation

```xml
<Synchronization type="Multi-Threaded">
  <Policy Name="Mutual Exclusion">
    <Adapts> FCFS </Adapts>
  </Policy>
  <Implementation Technique="Semaphore ">
</Synchronization>
```

3.3.2.3. Synchronization Technique Adaptation

As discussed earlier in this section, a service function can adapt its synchronization implementation technique. Let $T_O$ be the original implementation technique for synchronization and $T_A$ be the adapted technique. Both $T_A$ and $T_O$ can have one of the values specified in the Synchronization catalog. As all the synchronization techniques are the basic constructs for providing mutual exclusion, any technique can adapt to any other technique by replacing the constructs. There will be no change in synchronization behavior for adapting from
one technique to other. Thus, the synchronization level of specification for an adaptive service function will be:

Table 3.22: Synchronization specification of a function supporting synchronization technique adaptation

<table>
<thead>
<tr>
<th>&lt;Synchronization type=&quot;##Type of Synchronization&quot;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Policy Name=&quot;##Name of the Policy&quot;&gt;</td>
</tr>
<tr>
<td>&lt;Implementation Technique=&quot;##Name of the implementation technique&quot;/&gt;</td>
</tr>
</tbody>
</table>
| <PreCondition>"##Logical Assertion for the pre-condition <PreCondition/>
| <PostCondition>"##Logical Assertion for the post-condition<PreCondition/>
| <Action> "##Action taken by the service </Action>
| <Invariant> "##Logical Assertion for the invariant</Invariant> |
| </Synchronization> |

On similar lines, if the classify function of classifier service, adapts its synchronization policy implementation to “Monitors” from semaphore, then its synchronization level specification is presented below:
Table 3.23: Synchronization specification of the classify function supporting synchronization technique adaptation

```xml
<Synchronization type="Multi-Threaded">
    <Policy Name="Mutual Exclusion">
        <Adapts>FCFS</Adapts>
    </Policy>
    <Policy>
        <Implementation Technique="Semaphore">
            <Adapts>Monitors</Adapts>
        </Implementation>
    </Policy>
</Synchronization>
```

3.3.2.4. QoS Adaptation

The QoS level of a service specification indicates the measure of the nonfunctional properties of that service. This level of specification is used by the clients to make a selection decision if there are multiple choices available for a given type of service with different QoS guarantees. Among services with similar functional properties, the knowledge of quality attributes often could be the most significant information. It also adds to the confidence level with which application developers can use the service. It helps in predicting the value of the QoS of service properties of the overall system using some composition and decomposition rules.

Different QoS parameters can be specified in the service specification depending on the domain of the service. Some of the QoS parameters are static and some are dynamic. Static parameters are the parameters whose value remains constant during run-time over different execution environments. On the other hand, value of dynamic parameters can change during run-time. A detailed description of these parameters can be found in [BRA02].
One or many QoS parameters are specified in the service specification or new application dependent QoS parameters can be added in the specification. Following format, based on contract specification suggested by [MUKH03], is used to specify QoS level of specification for a function of the service.

Table 3.24: QoS specification of a function

```xml
<QoS>
  <Parameter Name="name of the parameter" Unit="Unit of measurement for this parameter" Value="Value of the parameter"/>
  # List of such parameters
</QoS>
```

For example, the QoS level specification for the classify function of the classifier service will be:

Table 3.25: QoS specification of the classify function

```xml
<QoS>
  <Parameter Name="Availability" Unit="percentage" Value="100"/>
  <Parameter Name="Turn-around-time" Unit="ms" Value="10"/>
  <Parameter Name="Accuracy" Unit="percentage" Value="30"/>
</QoS>
```

Following section identifies the impact of adaptation on QoS level of service specification. It also suggests new constructs added to the adaptive specification for representing the QoS of an adaptive service.
Adaptive QoS

As seen from above section, a service provides functions which may have dynamic or static QoS parameters. Values of static parameters are constant; hence without the loss of generality this thesis considers adaptation only of the dynamic QoS parameters.

While designing adaptive specification for the QoS level, there two important issues are considered. First, by default an adaptive service can provide different values for a QoS parameter. For example, the adaptive classifier service can support different types of classification algorithms which will have their respective turn-around-time values. Secondly, due to the adaptive nature of a service the QoS of a function can change. As discussed in earlier sections, there is a $\delta$, i.e., transformation, associated with each adaptation type. This $\delta$ affects different QoS parameters of the function such as the turn-around-time, precision, and performance. The overall effect on a QoS parameter due to adaptive behavior of a service is the sum of the effects on this QoS parameter due to all the $\delta$s. Thus, due to these two factors, the run-time value of any dynamic QoS parameter lies between a range. The adaptive specification for QoS level should represent this range of values for each dynamic parameter.

The QoS level of an adaptive specification for a function of the service is indicated as:
Table 3.26: QoS specification of an adaptive function

```
<QoS>
    <Parameter Name="##name of the parameter" Unit="##Unit of measurement for this parameter">
        <Value From="##lowest possible value of this parameter" To="Highest possible value of this parameter"/>
    </Parameter>
    ##List of such parameters
</QoS>
```

Thus, “To” and “From” attributes indicate the range of possible values for this parameter. For a static QoS parameter, both these values can be same, indicating the only possible value for this parameter.

The QoS level specification for classify function of the adaptive classifier service will be:

```
<QoS>
    <Parameter Name="Availability" Unit="percentage">
        <Value From="100" To="100"/>
    </Parameter>
    <Parameter Name="Turn-around-time" Unit="ms">
        <Value From="3" To="120"/>
    </Parameter>
</QoS>
```

This specification indicates that due to different adaptations supported at syntax, semantics, synchronization level and due to adaptations performed internally
(change in internal algorithms) by the classifier service its turn-around time ranges from 3 ms to 120 ms.

3.3.2.5. Adaptation by Learning Techniques
As described in section 3.1, this thesis considers a special type of adaptation which uses learning techniques, while designing the adaptive multi-level specification. This type of adaptation affects the contract of the service and hence, there is a need to consider the effects of learning technique on multi-level specification. As defined by Herbert Simon [SIMON83], learning is, any change in the system that allows it to perform better the second time on repetition of the same task or on another task drawn from the same population. For example, a classifier service can learn its list of acquaintance to give better classification result. This section identifies the impact of learning techniques in an adaptive service.

This thesis considers, without loss of generality, that one learning technique is associated with a function or method of the service. Thus, a service can contain different functions implementing different learning techniques. These learning algorithms affect the multi-level specification of the function. The impact of learning techniques on the multi-level specification is two-fold. First, as learning algorithms try to optimize certain non-functional attributes of the function, (e.g., turn-around-time, precision, performance, accuracy, etc.) it directly affects the QoS level of specification. Second, in some learning algorithms, either a feedback is expected from the clients in terms of reward/penalty or initial data set is expected from the clients for initiating learning data. This extra interaction demands addition of new functions in the adaptive specification. Following sections discuss both these aspects in detail.

Effects of Learning on QoS level of Specification
As discussed earlier, learning algorithms are used in a service to optimize its non-functional parameters. With each function call (interaction), the service learns with its experience and changes its functionality accordingly. Thus, a learning function converges to optimized value of QoS during run-time. Let the QoS parameters which are learnt by the learning algorithms be called as “Learnt Parameters”. Hence, during the lifetime of a service the “Learnt Parameters” of a function assume values from a specific range. This range should be considered while specifying the value for this parameter in the specification. During the service run-time, the actual value of the “Learnt parameter” will change according to the number of interactions. In the initial phases of learning algorithm, this value will be far away from the optimal value and eventually, it will converge to the optimal value. At any point of time, when a client needs to make a call to this learning function, it should be informed about which phase of learning the function is in. Thus, there should be an Indicator attribute associated with each “learnt parameters”, which represents the current value of that parameter. The QoS level of specification for a learning function \( f \) will be specified as:

**Table 3.28: QoS specification of a function implementing a learning technique**

```
<QoS>
    <Parameter Name="##name of the Learnt parameter"
        Unit="##Unit of measurement for this parameter">
        <Value From="##lowest possible value of this parameter" To="Highest possible value of this parameter"/>
        <Indicator Current="##Current value of the parameter"/>
    </Parameter>
#List of other parameters
</QoS>
```
This indicator parameter is a dynamic attribute in the adaptive service specification. The value of the current attribute changes as learning algorithm converges. Hence the QoS specification for the classify function which uses reinforcement learning technique to improve its acquaintance list is:

Table 3.29: QoS specification of the classify function implementing reinforcement learning technique

```
<QoS>
  <Parameter Name="Accuracy" Unit="percentage">
    <Value From="10" To="100"/>
    <Indicator Current="30"/>
  </Parameter>
</QoS>
```

Along with this Indicator attribute of the learnt parameters, a learning service may want to indicate properties associated with the learning algorithm of the function. This section identifies all such important properties a learning function can represent in its specification. As these properties of the learning technique are non-functional attributes of the function, they have to be specified in the QoS level of function specification. Following are the different parameters of the learning technique to be considered.

1. Type: This parameter specifies the type of learning technique used by the service. Value of this parameter can be reinforcement learning, supervised learning or unsupervised learning.

2. Feedback: Reinforcement learning techniques use feedback mechanism to assign rewards and penalties to the actions taken by the function. If this feedback is required to be given by clients then this parameter has to appear in the QoS of the function. This parameter can be either "Reward" or
“Penalty”. In case of “Reward” type feedback, a positive reward indicates success and negative reward indicates failure. On the other hand, for “Penalty” feedback type, positive penalty indicates failure and negative penalty indicates success.

3. Initial Data Set: Supervised or unsupervised learning uses initial data set as a learning data. If clients are required to provide this data then this parameter must appear in QoS specification of the function. This Initial data parameter has attributes such as the type, label, range, unit associated with it. This data set also has quality of service attributes such as the noise level and precision.

4. Learning Latency: All types of learning algorithms need a number of iterations to converge to optimal performance. The Learning latency for a learning algorithm is the number of iteration it takes to converge to optimal performance.

Based on these attributes, the QoS level of specification of an adaptive learning function of a service will be:
The Feedback and InitialDataset are optional parameters. They can be specified only when either the feedback or initial data set is provided by the client.

Hence the QoS specification for the classify function which uses reinforcement learning technique to improve its acquaintance list is:

```
<QoS>
  ##List of QoS parameters
  <Learning>
    <LearningType>##Type of learning</LearningType>
    <Feedback> ##Reward or Penalty</Feedback>
    <InitialDataSet>
      <NoiseLevel Unit="##unit of measurement" Value="##value of noise level"/>
      ##Other parameters concerning the initial data
      <Data Name="##Name of Data" Type="##Type of Data" unit="##unit of measurement of data" range="##range of data values" ##other attributes of the data/>
    </InitialDataSet>
    <LearningLatency Value="##value of learning latency" Unit="unit of measurement of learning latency"/>
  </Learning>
</QoS>
```
The classify function uses reinforcement learning and hence does not need any initial dataset from the clients.

Effect of feedback and Initial Data Set

Previous section described that learning algorithms need feedback and/or initial data set from clients. In case of reinforcement learning function, after every function call the service may require clients to provide feedback about the success of the call. For clients to provide a feedback, a new function has to be provided for every learning function in the service. Similarly, a learning function may need initial data set from the clients before any calls to the function are made. To provide this dataset, service needs to provide an init function for each such learning function. Thus, for every learning function \textit{fun} in the service which needs feedback from the clients following function has to be added to the adaptive specification:

```xml
<QoS>

##List of QoS parameters

<Learning>

<LearningType>Reinforcement</LearningType>

<Feedback>Reward</Feedback>

<LearningLatency Value="10" Unit="number of interactions"/>

</Learning>

</QoS>
```
Similarly, for every learning function fun in the service which needs initial data from the clients following function has to be added to the adaptive specification:

**Table 3.33: Specification for init function associated with a learning technique**

```xml
<Function Name="fun_Init">
  ## Syntax level of the init function
  ## Semantics of the init function
  ## Synchronization policy of the init function
  ## QoS of init function
</Function>
```

Hence, specification of the feedback function associated with the classify function of the classifier service is:
Table 3.34: Specification for feedback function associated with the classify function

```xml
<Function Name="fun_Feedback">
    ## Syntax level of the feedback function
    ## Semantics of the feedback function
    ## Synchronization policy of the feedback function
    ## QoS of feedback
</Function>
```

Summarizing, this chapter, MSAS provides a detailed design of different constructs for representing an adaptive service. This thesis considers adaptations at every level of specification and extends UMM specification for adaptive services. Thus, considering all the constructs for specifying an adaptive service in this chapter, Table 3.35 presents the complete MSAS in XML format of the classifier service.

Table 3.35: Complete Multi-level specification of the classifier service

```xml
<Service Name="Classifier">
    <Description> This service is a Information Classifier </Description>
    <InherentAttributes>
        <Author> Sucheta Phatak </Author>
        <Version>1.0 </Version>
        <DateDeployed> 2002-09-24 </DateDeployed>
        <ExecutionEnvironment> N/A </ExecutionEnvironment>
        <ComponentModel> N/A </ComponentModel>
        <Validity> 2010-06-17 </Validity>
        <Structure> N/A </Structure>
        <registrations> N/A </registrations>
    </InherentAttributes>
</Service>
```
<FunctionalAttributes>
  <TaskDescription>Classifies documents</TaskDescription>
  <AlgorithmandComplexity>N/A</AlgorithmandComplexity>
  <Alternatives>N/A</Alternatives>
  <Resources>
    <Architecture>N/A</Architecture>
  </Resources>
  <Designpatterns>N/A</Designpatterns>
  <Usages>Information Classification Systems</Usages>
  <Aliases>InformationClassifier</Aliases>
</FunctionalAttributes>

<FunctionsAndContract>
  <Function Name="Classify">
    <Syntax>
      <Adapts>Sort; Assort</Adapts>
      <Return Name="result" Type="DocumentClass">
        <Adapts>String</Adapts>
      </Return>
      <Arguments>
        <Adapts order="true"/>
        <Argument Name="doc" Type="Document">
          <Adapts>URL</Adapts>
        </Argument>
        <Argument Name="thesaurus" Type="String" Default="NetworkThesaurus.txt"/>
        <Argument Name="return_Type" Type="int" Default="0"/>
        <Argument Name="postProc" Type="int"/>
<Arguments/>

<Exceptions>RemoteException;FileNotFoundException; UnadaptiveException</Exceptions>

</Syntax>

<Semantics>

<PreCondition Expression="doc!=null &&
doc.getClass().toString().equals("Document") &&
doc.getType().equals("Text")">

<Adapts>doc!=null &&
doc.getClass().toString().equals("Document") &&
(doc.getType().equals("Text") ||
 (doc.getType().equals("MS Word Doc")))</Adapts>

</PreCondition>

<PostCondition Expression="result!= null">

<Adapts>result!= null &&
!result.getDocumentClass.IsEmpty()</Adapts>

</PostCondition>

</Semantics>

<Synchronization type="Multi-Threaded">

<Policy Name="Mutual Exclusion">

<Adapts>FCFS</Adapts>

</Policy>

<Implementation Technique="Semaphore">

<Adapts>Monitors</Adapts>

</Implementation>

</Synchronization>
<QoS>
  <Parameter Name="Availability" Unit="percentage">
    <Value From="100" To="100"/>
  </Parameter>
  <Parameter Name="Turn-around-time" Unit="ms">
    <Value From="3" To="130"/>
  </Parameter>
  <Parameter Name="Accuracy" Unit="percentage">
    <Value From="10" To="100"/>
    <Indicator Current="30"/>
  </Parameter>
</QoS>

<Learning>
  <LearningType>Reinforcement</LearningType>
  <Feedback>Reward</Feedback>
  <LearningLatency Value="10" Unit="number of interactions"/>
</Learning>

<Function Name="Classify_Feedback">
  <Syntax>
    <Return Type="void"/>
    <Arguments>
      <Argument Name="Reward" Type="int"/>
    </Arguments>
  </Syntax>
  <Semantics>
    <PreCondition
      Expression="this.prev_call.equals("Classify")"/>
    <PostCondition
      Expression="true"/>
  </Semantics>
</Function>
Expression="this.reward.registered==true" />
   
   <Invariant> this.newClassifyCall == false</Invariant>

</Semantics>

<Synchronization type="Single-Threaded"/>

<QoS>
   
   <Parameter Name="Availability" Unit="percentage">
      <Value From="100" To="100">
   </Parameter>

   <Parameter Name="Turn-around-time" Unit="ms">
      <Value From="2" To="2">
   </Parameter>

</QoS>

</Function>

<Function Name="Classify_init">

<Syntax>
   
   <Return Type="void"/>

   <Arguments>
      
      <Argument Name="policy" Type="int" />
   </Arguments>

   <Exceptions> UnadaptiveException </Exceptions>

</Syntax>

<Semantics>
   
   <PreCondition Expression="policy < 2 && policy > -1" />

   <PostCondition Expression="this.sync_policy != null" />

   <Invariant> true</Invariant>

</Semantics>

<Synchronization type="Multi-Threaded">
   
   <Policy Name="ReaderWriter"/>

</Synchronization>
Conclusively, this chapter provides a formal representation of a multi-level specification for an adaptive service. It identifies and defines different constructs for representing adaptations at syntax, semantics, synchronization and QoS levels of specification. Considering all these new constructs, a detailed XML schema of the multi-level specification (XML) is given in the Appendix. Thus, the MSAS forms the blue-print of an adaptive service which mapped on to an Object oriented language. To illustrate this, the thesis implements a translator for the multi-level specification to Java. Next chapter discusses this translator and its underlying mapping technique in detail.
CHAPTER 4. IMPLEMENTATION

Once a multi-level adaptive specification, discussed in the previous chapter, is created it serves two purposes, one as a blueprint of the service for the use by the outside world and as a design template of the service for the developer to implement that service. To assist the developer in this process, this thesis provides, TraMS, a translator which parses, validates and maps the MSAS of a service to corresponding Java language stubs. Because of the generic nature of the specification, this translator can be created to produce stubs in any Object Oriented Programming language. This section provides the design for mapping MSAS to JAVA language. Following sections describe the level by level mapping for MSAS to JAVA constructs. This mapping is based on the mapping suggested by [VAM06].

4.1. Service Attributes
As discussed in previous chapters, a multi-level service specification contains Inherent attributes and functional attributes. These attributes are considered static with respect to adaptation of the service, as they do not change during the adaptive behavior of the service. Hence, amongst the alternatives suggested by [VAM06] the TraMS chooses the approach of converting these attributes to block of comments in the stubs. This comment block will be between delimiters /* and */ and starting with a marker '@<Name of the Attribute>'. All the attributes having value "None" or "N/A" are not converted into comments. For instance, the inherent attributes of a classifier service, discussed in the previous chapter, will map on to following block of comments:
Table 4.1: Comment block for the Inherent Attributes of the classifier service

/* @InherentAttributes
Classifier’s Author is Sucheta Phatak
Classifier’s Version is 1.0
Classifier’s Date of Deployment is 2002-09-24
Classifier is valid till 2010-06-17
*/

4.2. Syntax Level Mapping

The TraMS converts the syntax specification of each function of a service into its corresponding definition in JAVA stubs. Considering, a specification of a service indicates its public functions, corresponding converted Java functions have the “public” access specifier. As indicated in the previous chapter, the Multilevel Specification of an adaptive service not only contains the syntax of each function but also the adaptations each function can perform. These adaptations, as described in the previous chapter, can be in form of Type adaptation, Name adaptation, adaptation in number and order of arguments. Hence while mapping specification functions into JAVA functions, these adaptations are considered.

Adaptations specified in the MSAS are such that the core functionality of the service does not change. Hence for each adaptive function of a service, the TraMS maps it to a Java function called as “<functionName>_core” in its skeleton. Using this skeletal code, the function is implemented by the service developer. The signature of the core function directly maps to base return type, base argument type and base name of the specification function. This core function expects the arguments in a specific order as indicated in the specification function and also, none of the base arguments are optional. Thus, the mapped core-function for the classify function of the classifier service is:
Table 4.2: Core-function skeleton for classify function of the classifier service

| Public DocumentClass Classify_core(Document doc, String Thesaurus) |
| { |
|     /*Core functionality*/ |
| } |

All the code for handling different types of adaptations supported by a function is placed into a wrapper function. A wrapper function associated with the function in the specification is the function which implements the code to handle adaptation and calls the core function. Following sections indicate the impact of different adaptations at the syntax level on the JAVA wrapper function.

4.2.1. Adaptation in order and number of arguments
As discussed in previous chapter, named parameters are considered for supporting the order adaptation and default values are considered for supporting number adaptation. JAVA does not provide named parameters, and hence, the TraMS uses string arrays containing name, type and value pairs to handle order adaptation. Thus, if a specification of a function indicates the order adaptation, while mapping this function to Java, its wrapper function’s arguments are expected as a String array from the clients. Every element of the String array contains, tuple of the form <Type;Name;Value> corresponding to each parameter. These parameters can be specified in any order. Clients can choose not to provide tuples for arguments which are declared as default. Java code to correctly convert this String array into arguments is inserted into the stubs by the TraMS. For instance, the wrapper function for classify function of the classifier service supporting adaptation in order and number of arguments is converted to the following java function:
Table 4.3: Wrapper function skeleton for classify function adapting in order of arguments

```java
public DocumentClass Classify(String[] args) {
    Document doc;
    String thesaurus = "defaultThes.txt";
    int return_Type = 0;
    int postproc = 0;
    //Get values of arguments from key-value pairs
    for(int i=0; i< args.length; i++) {
        String NTV[] = args[i].split(";");
        if(NTV[1].equals("doc")) {
            if(NTV[0].equals("URL")) {
                URL doc_a = new URL(NTV[2]);
                /*TODO Code to convert doc_a to doc*/
            }
        } else if(NTV[1].equals("thesaurus")) {
            thesaurus = NTV[2];
        } else if(NTV[1].equals("return_Type")) {
            return_Type = Integer.parseInt(NTV[2]);
        } else if(NTV[1].equals("postproc")) {
            postproc = Integer.parseInt(NTV[2]);
        }
    }
}
```
As JAVA does not support default arguments, if a function adapts only in the number of arguments, wrapper function for this function is overloaded. As clients can choose not to provide the trailing default arguments, wrapper function is overloaded by simpler versions of the wrapper function with fewer parameters. The main wrapper function accepts all the arguments. Overloaded functions of this wrapper function are provided with the trailing default parameters dropped. For instance, consider a dummy function:

<table>
<thead>
<tr>
<th>Table 4.4: Dummy function signature with some default arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo(&lt;T&gt; A, &lt;T&gt; B, default &lt;T&gt; C, default &lt;T&gt; D, default &lt;T&gt; e)</td>
</tr>
</tbody>
</table>

The main wrapper function will be:

<table>
<thead>
<tr>
<th>Table 4.5: Main wrapper function for Dummy function</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo(&lt;T&gt; A, &lt;T&gt; B, &lt;T&gt; C, &lt;T&gt; D, defult &lt;T&gt; e)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4.6: Overloaded versions of wrapper function</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo(&lt;T&gt; A, &lt;T&gt; B, &lt;T&gt; C, &lt;T&gt; D)</td>
</tr>
<tr>
<td>void foo(&lt;T&gt; A, &lt;T&gt; B, &lt;T&gt; C)</td>
</tr>
<tr>
<td>void foo(&lt;T&gt; A, &lt;T&gt; B)</td>
</tr>
</tbody>
</table>
Each of these overloaded wrapper functions will internally call the other wrapper
functions by specifying the default value of the dropped parameter. Thus, if the
classify function does not adapt in order of arguments but adapts in the number
of arguments, then its wrapper function is mapped as:

Table 4.7: Wrapper functions skeleton for classify function adapting in number of
arguments

```java
public DocumentClass Classify(Document doc, String Thesaurus, int return_Type, int postproc)  
{  
    /* code for classify wrapper */  
}
Public DocumentClass Classify(Document doc, String Thesaurus, int return_Type)  
{  
    int postproc = 0;  
    return Classify(doc, Thesaurus, return_Type, postproc);  
}
Public DocumentClass Classify(Document doc, String Thesaurus)  
{  
    int return_Type = 0;  
    Classify(doc, Thesaurus, return_Type);  
}
public DocumentClass Classify(Document doc)  
{  
    String Thesaurus = /* Default value of Thesaurus */;  
    return Classify(doc, Thesaurus);  
}
```
The return_Type and postproc arguments accepted by the Classify function in Table 4.7 are explained in 4.2.2.1 and 4.3.2.

4.2.2. Type Adaptation
In this type of adaptation, as discussed in the previous chapter, either the return type adapts or one or more argument types adapt. Following subsections discuss these adaptations in more details.

4.2.2.1. Adaptive Return Type
If a function supports the return type adaptation, then the original return type given in the specification is called “base type” and types to which it adapts are called as adaptive types. The return type of the mapped function can be any of the adaptive types or the base type. Hence, the TraMS maps the return type to Java “Object” type. As all Java types are inherited from “Object” class this mapping is appropriate. For a Java function to return the required type, clients need to specify their requirement at run-time. Hence, the TraMS maps a function supporting adaptive return type to a corresponding java function that takes an extra argument, which specifies the required return type. This argument is of type “int” with Name="return_Type" and has a default value 0, indicating base return type. For every adaptive return type, R\textsubscript{1}, R\textsubscript{2},…..R\textsubscript{n}, the value of return_Type argument is 1,2,…..n. This mapping of return type adaptation requires a change in the MSAS of the service. The syntax level of a function is updated to add this new argument for the function. This argument is a non-adaptive argument but is subject to change its order in the arguments list. Thus, the syntax level of the MSAS of the classify function of the adaptive classifier service adapting in its return type is:
### Table 4.8: MSAS of the classify function supporting adaptive return type

```xml
<Function Name="Classify">
  ....
  <Return Type="Classification" Name="result">
    <Adapts>String </Adapts>
  </Return>
  <Arguments>
    <Argument Type="Document" Name="doc">
      <Adapts>URL </Adapts>
    </Argument>
    <Argument Type="String" Name="return_Type"/>
  </Arguments>
</Function>
```

This specification gets converted to following function:

### Table 4.9: Wrapper function for the classify function supporting adaptive return type

```java
public Object Classify(Document doc, int return_Type)
{
    Object return_result = null;
    DocumentClass result = Classify_core(doc);
    if(return_Type==1)
    {
        return_result = result.toString();
    }
    else
    {
        return_result = result;
    }
    return return_result;
}
```
4.2.2.2. Adaptive Argument Type

If a function supports argument type adaptation, then there are two cases to be considered for mapping this adaptation to corresponding Java functions. The first case is if the same function supports order adaptation then the mapped Java function signature will have String array as argument. The function will convert the argument tuples to correct type required by the clients. Skeleton of the conversion code is added by the TraMS in the Java stubs. On the other hand, if the function does not support the order adaptation, then the function arguments are directly mapped to the Java function arguments. All the adaptive types are mapped to Java “Object” type to support either the adaptive types or the base type. Then the function will internally convert the given type to required type. Skeleton of the code for converting these types is provided by translator. The actual conversion code for primitive types like String, int, float is provided by the translator. On the other hand conversion code for user defined types has to be implemented by the service developer.

Thus the Java wrapper function for classify function which supports adaptive argument types and does not support order adaptation is given in Table 4.10.
4.2.3. Name Adaptation

As indicated earlier, a function of an adaptive service can adapt its name which is specified in MSAS of the service. For mapping such a function to Java stub function, the TraMS maps each adaptive name to a new function with the adaptive name but same signature as the base (original) function. This new
function is a dummy function which calls the base function directly. The signature of this dummy function exactly matches the signature of the base function. Hence, if the base function provides any other type of adaptations which modify its signature, then it indirectly affects the dummy function’s signature.

Thus, the classify function with adaptive names “Sort” and “Assort” gets mapped to following dummy functions:

Table 4.11: Dummy functions for the classify function supporting adaptive names

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>public Object Sort(Object doc, int return_Type)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>return Classify(doc,return_Type);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>public Object Assort(Object doc, int return_Type)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>return Classify(doc,return_Type);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

4.3. Semantic Level Mapping

The semantic level of the MSAS of a service, as discussed earlier, specifies pre-conditions, post-conditions and invariants for every function of the service and their respective adaptations. The TraMS considers the approach of mapping these logical expressions to Java assert construct [J2SE101]. In [VAM06], different constructs are suggested for pre/post conditions depending upon its type. All these necessary constructs for the pre-condition are placed in the body of the function as the first statement so that they can be evaluated before the functions starts executing. Similarly, all the necessary constructs for the post-condition are placed in the body of the function as the last statement so that they
can be evaluated before the method starts executing. Likewise, the invariant of a function is mapped on to assert statements [J2SE101] that verify the logical expression associated with the invariant. This assertion is evaluated at the start of the function, just after the pre-condition is evaluated and at the end of the function, just after the post-condition is evaluated. For mapping the adaptive pre and post conditions TraMS uses the wrapper Java function corresponding to the specification function. The wrapper java function implements the code for handling the pre and post processing. The mapped Java core function associated with this specification function only checks for the base pre, post conditions and invariants. The TraMS automatically adds the assertions corresponding to base pre-condition, post-condition and invariant in the skeleton of the core function. For instance the skeleton generated by TraMS for core function associated with the classify function is:
Table 4.12: Core-function skeleton for classify function of the classifier service with assertions added by TraMS

```java
Public DocumentClass Classify_core(Document doc, String Thesaurus)
{
    assert doc!=null &&
    doc.getClass().toString().equals("Document") &&
    doc.getType().equals("Text"): "Pre-Condition for Classify Violated";
    assert this.ACQ!=null: “Invariant for Classify violated”;

    /* Core functionality */
    assert this.ACQ!=null: “Invariant for Classify violated”;

    assert result!= null &&
    !result.getDocumentClass.IsEmpty():
        “Post-Condition for Classify violated”;
}
```

The wrapper JAVA function is such that it supports any of the adaptive pre-conditions and ensures any of the adaptive post-conditions required by the clients. The TraMS adds the code to achieve this feature in the skeleton of the wrapper Java function. Following sections indicate the required mapping for adaptive pre and post conditions.

4.3.1. Adaptive PreCondition

As described in chapter 3, the adaptive pre-conditions demand pre-processing activity before the core functionality is executed. This pre-processing corresponds to the \( \delta_{Pre} \) which denotes the transformation of the adaptive service.
For every adaptive pre-condition of the function, there is a $\delta_{\text{Pre}}$ associated. The TraMS maps these adaptive pre-conditions to conditional checks using if constructs. Thus, for each adaptive pre-condition following code is added by the TraMS to generated skeleton of the wrapper function.

**Table 4.13: Code added to skeleton of wrapper function to check adaptive pre-conditions**

<table>
<thead>
<tr>
<th>Code added to skeleton of wrapper function to check adaptive pre-conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>if( &lt;Logical Expression of the adaptive pre-condition(i)&gt; )</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>/* Code performing $\delta_{\text{Pre}}$ associated with this adaptive pre-condition */</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Accordingly, the code generated by the TraMS for adaptive pre-condition of for classify function is:

**Table 4.14: Wrapper function skeleton for classify function of the classifier service with conditional checks added by TraMS for adaptive pre-conditions**

```java
public Object Classify(DocumentClass doc, int return_Type)
{
    /* Code handling syntax adaptation */
    if(doc!=null &&
    doc.getClass().toString().equals("Document") &&
    (doc.getType().equals("Text")||
    doc.getType().equals("MS Word Doc")))
    {
        /* Code to transform the state to base pre-condition:
        doc!=null &&
        doc.getClass().toString().equals("Document") &&
        doc.getType().equals("Text") */
```
if(doc.getType().equals("MS Word Doc"))
{
    try
    {
        POIFSFileSystem fs = new
        POIFSFileSystem(new FileInputStream(doc.getpath()));
        HWPFDocument wdoc = new HWPFDocument(fs);
        WordExtractor we = new WordExtractor(wdoc);
        BufferedWriter out = new BufferedWriter(new
        FileWriter(doc.getName()+".txt");
        out.write(we.getText());
        out.close();
        doc.setpath(doc.getName()+".txt");
        doc.setType("Text");
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
}
Classify_core(doc);

4.3.2. Adaptive PostCondition
For a function, adaptive post-conditions demand post-processing activity after the
core functionality is executed. This post-processing, as described in the 3rd
chapter, corresponds to the $\delta_{Post}$ which denotes the transformation of the
adaptive service. For every adaptive post-condition of the function, there is an
associated $\delta_{Post}$. As opposed to pre-conditions, post-conditions are the guards
that the function ensures. Thus, if clients want the function to ensure an adaptive post-condition than the base post-condition, then this has to be explicitly indicated by the clients at run-time. As there can be many such adaptive post-conditions to which a function adapts, the TraMS requires the function to support a new argument using which clients can specify there required post-condition. This argument is of type Java “int” with Name=“postproc” and has a default value “0” indicating the base post-condition. For specifying one of the adaptive post-conditions $P_{A1}$, $P_{A2}$, …..$P_{An}$, the value of this argument should be set to 1,2……n respectively. Addition of this new argument requires a change in the MSAS for the service. The syntax level of the function is updated to add this new argument for the function. This argument is a non-adaptive argument but is subject to change its order in the arguments list.

For every adaptive post-condition, the TraMS adds following code after the core functionality of the function is executed.

Table 4.15: Code added to skeleton of wrapper function to check adaptive post-conditions

```
if( postproc == <i>)
{

    /* Code performing $\delta_{Post<i>}$ associated with this adaptive post-condition <i> */
    assert <adaptive post-Condition <i>>: “Post-Condition for <Function_Name> Violated”;
}
```

Accordingly the code generated by TraMS for handling adaptive post-condition of classify function is:
public Object Classify(DocumentClass doc, int return_Type, int postproc)
{
    /* Code handling syntax adaptation */
    /* Code handling adaptive pre-condition */
    /* call to Classify_core */
    if( postproc == 1)
    {
        /* Code to transform to 
        result!= null &&
        !result.getDocumentClassIsEmpty() */

        DocumentClass result_temp =
        (DocumentClass)result;
        if(result_temp.getDocumentClass().isEmpty())
            result_temp.setDocumentClass("Default");

        assert result_temp!= null &&
        !result_temp.getDocumentClassIsEmpty(): “Post-Condition
        for Classify violated”;
    }
}
synchronization technique is implemented and hence, no synchronization level mapping is needed. Multi-threaded services need to implement synchronization mechanism depending on the service domain. As discussed in chapter 3, a standard list of basic synchronization policies a function can use, is provided by policy catalog [ANJ04]. A detailed description of all these policies along with the description of their behavior is catalogued in [ANJ04].

The TraMS considers the mapping of only basic policies, indicated below, to Java constructs. However, a similar approach that is suggested for these policies can be followed for other policies in the catalog. A detailed description of these policies is given below:

**Mutual Exclusion Policy:** This policy is used for ensuring mutual exclusion for executing critical sections. If thread $T_i$ is executing its critical section then no other thread can execute its critical section. Mapping a function supporting this policy, depends on the technique of synchronization used. One of the following different techniques can be used to achieve mutual exclusion policy.

- **Semaphores:** A function implementing mutual exclusion using semaphores is mapped to a java function which uses `java.util.concurrent.Semaphore` class.

- **Monitors:** A function implementing mutual exclusion using monitor is mapped to a “synchronized” java function.

- **Locks:** A function implementing mutual exclusion using locks is mapped to a java function which uses `java.util.concurrent.locks.Lock` class.

- **Conditional locks:** A function implementing mutual exclusion using conditional locks is mapped to a java function which uses `java.util.concurrent.locks.Lock` and `java.util.concurrent.locks.Condition` classes.
**Barrier:** This synchronization policy allows multiple threads to wait for each other after finishing execution of their respective critical sections. The TraMS maps this policy to `java.util.concurrent.CyclicBarrier` class irrespective of the technique indicated by specification of the service. This is because the `CyclicBarrier` construct provided by `java.util.concurrent` does not allow users to specify the internal construct used to achieve this policy. If the developer wants to use a specific technique, like semaphore, he/she will have to implement the policy from scratch.

**Bounded buffer:** This policy, also called as Producer-Consumer, is useful where a buffer in shared memory is used to transfer information from one thread (the producer) to another thread (the consumer). The TraMS maps this policy to `java.util.concurrent.SynchronousQueue<E>` class irrespective of the technique indicated by specification of the service.

**ReaderWriter:** This policy deals with situations where multiple reading threads can access shared memory simultaneously. On the other hand writing threads have exclusive access to shared memory. The TraMS maps this policy to `java.util.concurrent.locks.ReadWriteLock` class irrespective of the technique used for the policy.

**FCFS Priority:** This is a priority based synchronization policy, where priority is assigned to tasks according to the arrival time, i.e., on a first come first serve basis. Tasks are executed according to their priority. The TraMS maps this policy to using `java.util.concurrent.locks.ReentrantLock` class created with `fair="true"` which is first-come-first-serve. In this case, if many lock requests are made at the same time, they are granted very close to the order in which they are made.

The service developer is required to implement these constructs appropriately according to the service functionality. The base policy of the function can be one
of the above policies. Depending on this base policy, a service can adapt to some of the policies indicated in the catalog. As described in chapter 3, a synchronization policy can adapt to a “Constrained” policy or to a “Relaxed” policy or to an “Equivalent” policy. Hence, the TraMS maps only some of the adaptive policies depending on the base policy. For each of the given policies following list indicates its respective permitted adaptive policies.

- Mutual Exclusion Policy: This base policy can adapt in its technique of implementation. A service providing a function with Mutual Exclusion policy using semaphores can adapt to Mutual Exclusion using monitors. Also, the Mutual Exclusion policy can adapt to the FCFS policy because the Mutual Exclusion policy is a “Relaxed” version of the FCFS policy.
- Barrier: The TraMS does not allow adaptation for this policy.
- Bounded buffer: The TraMS does not allow adaptation for this policy.
- ReaderWriter: The TraMS does not allow adaptation for this policy.
- FCFS Priority: This policy can only adapt to Mutual Exclusion policy, because FCFS is a “Constrained” version of the Mutual Exclusion policy.

The TraMS does not allow adaptation for Barrier, Bounded buffer and ReaderWriter policy because these policies implement specific and independent synchronization behavior. Adapting these techniques would indicate changing the core synchronization behavior of the service. Hence an adaptive service should use a better design technique like inheritance, rather than adapting these policies.

Thus, for a function of a service which provides an adaptive policy the TraMS maps an _init function with name as, <FunctionName>_init. This _init function needs to be executed before any calls to the respective function are made. This _init function takes an integer argument indicating which adaptive synchronization policy is to be used. The specification of this _init function has to be added to the MSAS of the service.
The default value of the synchronization policy variable of the service is set to zero, indicating it to be the base synchronization policy. Every adaptive synchronization policy and adaptive synchronization technique in the specification has an associated value of synchronization policy variable. Adaptive synchronization techniques, ST\textsubscript{A1}, ST\textsubscript{A2}, ST\textsubscript{A3}......ST\textsubscript{Am}, represented in this order in the specification, are denoted by values 1,2,3...m respectively. Adaptive synchronization policies, SP\textsubscript{A1}, SP\textsubscript{A2}, SP\textsubscript{A3}......SP\textsubscript{An}, represented in this order in the specification, are denoted by values m+1,m+2,m+3......m+n respectively. This synchronization policy variable is mapped as a private member of the Java class representing the service with the name “sync\_policy”. Due to the multi-threaded nature of the service, proper synchronization has to be provided for accessing this sync\_policy variable. The init function updates this variable and the wrapper function reads it. Hence, TraMS provides a ReadWrite lock for accessing the sync\_policy variable. It adds the necessary lock and release code in the init and wrapper functions for this lock. The init function’s corresponding Java function in the skeleton code generated by the TraMS is:

Table 4.17: init function skeleton for a function supporting Synchronization policy adaptation

<table>
<thead>
<tr>
<th>public void &lt;functionName&gt;_init(int policy) throws UnadaptiveException</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
</tr>
<tr>
<td>if(!this.syncPolicylock.writeLock().tryLock())</td>
</tr>
<tr>
<td>throw new UnadaptiveException();</td>
</tr>
<tr>
<td>this.synch_policy  = policy;</td>
</tr>
<tr>
<td>this.syncPolicylock.writeLock().unlock();</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
The init function shown in Table 4.17, tries to get a write lock for the synch_policy variable. If any thread currently has a read lock on the synch_policy i.e. at least one thread is executing the wrapper function, then the init function throws UnadaptiveException, indicating failure to adapt.

As described in the earlier section, a specification function’s core functionality is implemented in <functionName>_core java function. If the specification function has adaptive synchronization policies and/or adaptive synchronization techniques, then for each of these policies and the base policy, a corresponding synchronization construct should be provided in the mapped function. TraMS provides the code in the skeleton of the wrapper function for defining these constructs according to the “sync_policy” variable. For monitor synchronization technique of mutual exclusion, TraMS maps a new “Synchronized Wrapper” java function. The outer wrapper function responsible for handling all other adaptations, calls the “Synchronized Wrapper” function if the “sync_policy” variable indicates mutual exclusion using monitors. For all other policies and techniques, TraMS adds the code in the wrapper function to define and instantiate respective java classes into synchronization objects. The service developer appropriately uses these synchronization objects before calling the core function.

For the classify function of the classifier service, which supports the Mutual Exclusion policy with monitors and adapts to the FCFS policy, the skeleton code generated by TraMS is:

```
Table 4.18: Skeleton code generated by TraMS for classify function of classifier service supporting synchronization policy adaptation

public void Classify_init(int policy) throws UnadaptiveException {
```

if(!this.syncPolicylock.writeLock().tryLock())
    throw new UnadaptiveException();
this.synch_policy = policy;
this.syncPolicylock.writeLock().unlock();
}

public Object Classify(Object doc, String return_Type, int postCondition)
{
    // Code handling other adaptations
    this.syncPolicylock.readLock().lock();
    if(this.sync_policy == 1)
    {
        ReentrantLock lock = new ReentrantLock(true);
        Lock.lock();
        try
        {
            // Call to Classify_core()
        }
        finally
        {
            lock.unlock();
        }
    }
    else if (this.sync_policy == 0)
    {
        Classify_core_Monitor();
    }
    this.syncPolicylock.readLock().unlock();
    // Code for handling adaptive post-conditions
public synchronized DocumentClass
Classify_core_Monitor(Document doc)
{
    // Call to Classify_Core;
}

4.5. QoS Mapping

This section considers QoS level adaptation for specification function and
mapping it onto corresponding Java functions. Dynamic QoS parameters of a
function can adapt their values at runtime. On the other hand, static QoS
parameters are constant and hence, are non adaptive. Hence, TraMS considers
only the mapping for dynamic QoS parameters. The TraMS maps the static QoS
parameters of a function to corresponding comment blocks. Each parameter is
mapped to a block of comments between delimiters /* and */. Thus, static QoS
parameters of classify function are specified as:

|Parameter Name="Adaptability" Unit="percentage">
|<Value From="70" To="70">
|</Parameter>

|Parameter Name="Maintainability" Unit="ms">
|<Value From="50" To="50">
|</Parameter>

These static parameters are mapped to:
As dynamic QoS parameters adapt their values at runtime, these values are within a range. To specify the correct range in the specification, service developer should first derive these values after the service has been implemented. TraMS assist the service developer in determining these values, by providing the code related to Dynamic QoS measurement in the function skeleton. This section identifies the Java mapping for technique of measurement of the dynamic QoS parameters specified in the catalog [BRA02]. TraMS maps each of these dynamic QoS parameter to a function variable with name="<Fun_Name>_<NameOfQoSParameter>", of type $T_Q$. The TraMS inserts the code to calculate the value associated with the QoS parameter using the variable into the function skeleton.

Ordering constraints – This parameter indicates whether the results returned by the function of a service are in the proper order. TraMS maps this parameter to a variable of type $T_Q = boolean$. TRAMS inserts the code in the mapped wrapper Java function to check the ordering of the return parameter of the core function and sets the variable accordingly.

Throughput – This parameter measures the number of requests a method can serve per unit time. TRAMS maps this parameter to a variable of type $T_Q = float$. TRAMS inserts the code in the mapped wrapper Java function to calculate the throughput using turn-around-time ($1/\text{turn-around-time}$) and sets the variable to that value.

Capacity – This parameter measure of the number of concurrent requests the component can serve at a given time instant. TRAMS maps this parameter to a variable of type $T_Q = int$. Functions with single-threaded synchronization type,
have a default value of "1.0" for this variable. The value of this parameter depends on the synchronization policy used by the function. For instance, mutual exclusion policy will allow only 1 request to be served at a given time, but ReaderWriter policy will allow multiple readers to execute the function at a given time. TRAMS inserts the code in the mapped synchronized wrapper Java function to increment the value of this variable, indicating the number of requests served by the function at that time.

Turn-around Time – This parameter measures the time taken by the method to return the result. TRAMS maps this parameter to a variable of type $T_Q = \text{float}$. TRAMS inserts the code in the mapped wrapper Java function to calculate the difference between the start-time of the call and the end-time of the call and sets the variable to that value.

Availability – This parameter indicates the duration when a function is available to offer a particular service. It is calculated in terms of the percentage of time the function is available to offer its services. For distributed services, this parameter indicates the availability of the function and not that of the network. This parameter can be calculated by using a feedback mechanism for the function. TraMS maps this parameter to a service variable type $T_Q = \text{float}$, indicating the function's availability in percentage. In the feedback function associated with this function, the variable is updated to indicate the availability using the feedback sent by the clients. This variable is calculated as, $(100/\text{number of times a request for the function was made}) \times (\text{the number of times a request was served})$.

Thus using the skeleton provided by the TRAMS, service developer can create a working implementation of an adaptive service. He/she can then perform black box testing of the service and determine the value range for each of the dynamic QoS parameters. Thus, once the range of values is determined, he can update the QoS specification.
Thus, Turn-around-time, of the classify function specified in Table 4.21 is mapped to a function variable “Classify_Turn-around-time”. Similarly, the availability is measured using service variable “Classify_availability”. Skeleton code inserted by TraMS for calculating the values of these variables is represented in Table 4.22.

Table 4.21: Dynamic QoS parameter specification for classify function

```
<Parameter Name="Availability" Unit="percentage">
   <Value From="A1" To="A2"/>
</Parameter>
<Parameter Name="Turn-around-time" Unit="ms">
   <Value From="T1" To="T2"/>
</Parameter>
```

Table 4.22: Dynamic QoS parameter mapping for classify function

```
public Object Classify(Object doc, String return_Type, int postCondition)
{
    float Classify_Turn-around-time =
        System.currentTimeMillis();
    //code to handle all adaptations and core functionality
    Classify_Turn-around-time = System.currentTimeMillis()
        -
        Classify_Turn-around-time;
}
public void Classify_Feedback(int reward)
{
    if(reward>0)
    {
        this.Classify_availability =
            ((this.Classify_numReq -1) * this.Classify_availability +
            100)/ this.Classify_numReq;
    }
    else
    {
        this.Classify_availability =
            ((this.Classify_numReq -1) * this.Classify_availability -
            100)/ this.Classify_numReq;
    }
}
```

The Table 4.22 shows turn-around-time being calculated by the Classify function and availability is measured using the feedback function. The variable
Classify_numReq indicates the total number of requests made to the classify function.

4.6. Mapping Learning Functions
As discussed in chapter 3, learning techniques can have an impact on the specification of learnt QoS parameters. This requires the learnt QoS parameter to specify “Indicator” attribute which is a dynamic feature. The “current” value of the “indicator” attribute changes during the learning phase. Therefore, the TraMS maps the learnt parameter’s indicator attribute to a private variable of the service named as `<functionName>_<LearntParameterName>_Current`. The TraMS creates a skeleton of a getter function for this variable of the form:

<table>
<thead>
<tr>
<th>Table 4.23: Skeleton code for getter function of Learnt Parameter</th>
</tr>
</thead>
</table>
| public String  
get<FunctionName>_<LearntParameterName>_Current()  
{  
    return  
this.<functionName>_<LearntParameterName>_Current;  
} |

The service developer implements the code to update the value of this indicator as the learning progresses. The QoS properties added to the specification for indicating the properties of learning algorithm are converted to comment blocks in the skeleton code.

Chapter 3 indicates the addition of a new specification function in the MSAS of a service which has a learning function. These feedback and init functions are considered as non adaptive functions and hence, they can be mapped directly to a Java function with corresponding arguments and return types. If a function provides a synchronization adaptation and also implements a learning algorithm
then the respective init functions are clubbed into one init function. Skeleton code generated for the feedback and the init function is specified in Table 4.24.

<table>
<thead>
<tr>
<th>Table 4.24: Skeleton code for feedback and init function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>public &lt;return_Type&gt; &lt;function_name&gt;_feedback (&lt;arguments&gt;)</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

### 4.7. Order of Mapping

This section discusses the order of mapping different adaptations of an adaptive function to the Java code. As discussed in this chapter, an adaptive service supports adaptation at different levels of the specification. Code handling these adaptations is inserted by TraMS into the corresponding Java stub of the service. There is a specific order in which this code is inserted into the Java stub. This section discusses this order in detail.

As proposed by Beugnard et al. [BEU99], specification of a service is divided into multiple levels for providing separation of concerns. This means that every level of specification represents different independent aspects of the service such as, syntax, semantics, synchronization and QoS. Although, these levels are independent, they represent features of the same service. Hence, while designing the order of mapping for different adaptations, TraMS assumes them to be independent of each other. Figure 4.1 indicates the order in which adaptations at different levels are mapped by TraMS. As presented in this figure, TraMS starts with mapping the order adaptation. The Java stubs generated
contain the code for handling order adaptation at the beginning. This is because before checking for any other adaptations the correct arguments have to be retrieved from the name-value pairs. Then the code for handling argument type adaptation is inserted, where the necessary type conversions are performed. The code for handling adaptation in number of arguments is in forms of overloaded functions and hence can be mapped in any order. Similarly the code handling name adaptation is using dependent functions, hence can be mapped in any order. After argument type adaptation, the code handling adaptive pre-conditions is inserted. Before calling the core functionality, the code handling synchronization adaptation is added to the stub. After the core functionality is executed, the code handling adaptive post-conditions is added. This is followed by the code handling the return type adaptation which performs the necessary transformation of the return type. The order of code measuring QoS parameter depends on the nature of the parameter. For example, code measuring the turn-around-time is added at the beginning of the function, before any other adaptation is handled. Also the code for the learning technique depends on the type of learning technique and the offline or online nature of the learning technique.
Summarizing the mapping provided by this chapter, Table 4.25 provides the complete skeleton code generated by the TraMS for the classifier service.

Table 4.25: Stub generated by TraMS for Adaptive Classifier Service

```java
public class Classifier {
    public Object Classify(String[] args) throws RemoteException, FileNotFoundException {
        float Classify_Turn-around-time = System.currentTimeMillis();
        Document doc;
    }
```
Object retrun_result = null;
String thesaurus = "defaultThes.txt";
int return_Type = 0;
int postproc = 0;

//Get values of arguments from key-value pairs
for(int i=0; i< args.length; i++)
{
    String NTV[] = args[i].split(";");
    if(NTV[1].equals("doc"))
    {
        if(NTV[0].equals("URL"))
        {
            URL doc_a = new URL(NTV[2]);
            /*TODO Code to convert doc_a to doc*/
        }
    }
    else if(NTV[1].equals("thesaurus"))
    {
        thesaurus = NTV[2];
    }
    else if(NTV[1].equals("return_Type"))
    {
        return_Type = Integer.parseInt(NTV[2]);
    }
    else if(NTV[1].equals("postproc"))
    {
        postproc = Integer.parseInt(NTV[2]);
    }
}
if(doc!=null &&
doc.getClass().toString().equals("Document") &&
(doc.getType().equals("Text") || (doc.getType().equals("Word Doc"))))
{
    /*TODO Code to transform to base pre-
    condition : doc!=null && doc.getClass().toString().equals("Document") && doc.getType().equals("Text")*/
}

if(this.synch_policy==0)
{
    Semaphore sem = new Semaphore(1);
    sem.acquire();
    DocumentClass result = Classify_Core(doc, thesaurus, return_Type, postproc);
    sem.release();
}
else if(this.synch_policy==1)
{
    DocumentClass result = Classify_Synchronized(doc, thesaurus, return_Type, postproc);
}
else if(this.synch_policy==2)
{
    ReentrantLock lock = new ReentrantLock(true);
    lock.lock();
    try
    {

DocumentClass result = 
Classify_Core(doc, thesaurus, return_Type, postproc);
}
finally
{
    lock.unlock();
}

if(postProc == 1)
{
    /* TODO Code to transform from base post-condition :result!= null To: result!= null &&
    !result.getDocumentClass.IsEmpty()*/
    assert (result!= null &&
    !result.getDocumentClass.IsEmpty()) : "Adaptive PostCondition Violated";
}

if(return_Type==1)
{
    /*TODO Code to Convert result to String*/
}

Classify_Turn-around-time = 
System.currentTimeMillis() - Classify_Turn-around-time;
return return_result;
}

public Object Sort(String[] args) throws 
RemoteException, FileNotFoundException
{ return Classify(args); }

public Object Assort (String[] args) throws RemoteException, FileNotFoundException {
    return Classify(args);
}

private synchronized DocumentClass Classify_Synchronized(Document doc, String thesaurus, String return_Type, int postproc) throws RemoteException, FileNotFoundException {
    DocumentClass result = Classify_Core(doc, thesaurus, return_Type, postproc);
    return result;
}

private DocumentClass Classify_core(Document doc, String thesaurus, String return_Type, int postproc) throws RemoteException, FileNotFoundException {
    assert (doc!=null &&
    doc.getClass().toString().equals( "Document") &&
    doc.getType().equals("Text")) :"PreCondition Violated";
    assert ( this.ACQ != Null ) :"Invariant Violated";
    /*TODO Core functionality*/
    assert ( this.ACQ != Null ) :"Invariant Violated";
    /*TODO Core functionality*/

}
assert (result!= null) : "PostCondition Violated";
}

public void Classify_Feedback(int reward)
{
    assert (this.prev_call.equals("Classify")): "PreCondition for Classify_Feedback violated";
    assert (this.newClassifyCall == false): "Invariant for Classify_Feedback violated";

    if (reward>0)
    {
        this.Classify_availability = ((this.Classify_numReq - 1) * this.Classify_availability + 100)/ this.Classify_numReq;
    }
    else
    {
        this.Classify_availability = ((this.Classify_numReq - 1) * this.Classify_availability - 100)/ this.Classify_numReq;
    }
    /* code to register the feedback */
    assert (this.newClassifyCall == false): "Invariant for Classify_Feedback violated";
    assert (this.reward.registered==true): "PostCondition for Classify_Feedback violated";
}

public void Classify_init(int policy) throws
UnadaptiveException
{
    assert(policy < 2 && policy > -1): "PreCondition for Classify_init violated";

    if(!this.syncPolicyLock.writeLock().tryLock())
        throw new UnadaptiveException();
    this.synch_policy = policy;
    this.syncPolicyLock.writeLock().unlock();
    assert(this.synch_policy!=null): "PostCondition for Classify_init violated";
}

Conclusively, this chapter provides the overall mapping technique used by TraMS in converting MSAS of a service to Java stubs. The chapter identifies Java mapping for all adaptation types at each level of specification. Thus, using TraMS a service developer can create Multi-level specification for an adaptive service and create the corresponding Java stub. These stubs will be updated by the service developer to provide an adaptive service implementation “designed for change”.
CHAPTER 5. TRANSLATOR IMPLEMENTATION AND CASE STUDY

As discussed in chapter 4, this thesis provides a translator that maps the Multi-level specification of an adaptive service to corresponding Java-based skeleton code. This chapter provides the detailed design and implementation of this translator called as TraMS. This chapter discusses the design of TraMS, and presents experiments done with it in the context of three adaptive services, Adaptive Classifier, Adaptive Tracking Service, and Adaptive Container. An empirical evaluation of the adaptive classifier service in terms of the lines of code, accuracy, and turn-around-time is also provided in this chapter.

5.1. Design of TraMS

TraMS is developed in Java and the GUI is developed using Windows Forms in .Net2.0. This section discusses the structural design of TraMS. As shown in Figure 5.1, the functionality of the TraMS involves following phases:

- Generating the MSAS.
- Parsing, Validating, and Analyzing the MSAS.
- Translating the MSAS to corresponding Java code of the stubs.

Hence, TraMS is composed of the GUI, the MS_Parser, and the MS_Translator, as shown in Figure 5.1. The GUI provides a front end for creating the adaptive service details by the user and converting it to a XML MSAS. The MS_Parser parses, validates, and analyzes this XML MSAS and converts it into an intermediate service representation. The MS_Translator then converts this intermediate service representation to Java stubs. Following sections discuss each of these modules in detail.
Figure 5.1: Structural Design of TraMS

Figure 5.2 below, indicates the class diagram of TraMS. As seen from the diagram, every module in TraMS is represented as a class. The GUI class is composed of a main GUI form which contains multiple MSAS GUI instances. This design enables creating and updating multiple service specifications simultaneously. MSAS_GUI class is responsible for getting the service related details from the user.
5.1.1. GUI

This module of the TraMS is developed as a Windows form application (.Net 2.0). It provides a tabbed view to the user for specifying the details of an adaptive service. The GUI module provides a front end for specifying the Inherent Attributes, Functional Attributes of the service and Functions and Contracts of every function of the service. A sample GUI screenshot is provided in Figure 5.3. Other screen shots of the GUI are included in the Appendix.
5.1.2. MS_Parser

This module of TraMS is responsible for parsing and validating the MSAS created using the GUI module. The grammar for the XML MSAS is specified as a XML schema, which validates the structure of the XML MSAS. MS_Parser also validates the contents of tags specified in the XML MSAS. For instance, the logical assertion specified for pre-condition or post-condition are validated. This module converts the XML MSAS into an intermediate service representation. The intermediate service representation is specified in terms of a class diagram in Figure 5.4. This class diagram indicates a service class to which a service specified in the MSAS is translated. This service class is a container class storing Inherent Attributes, Functional Attributes and functions of a service. The Function
class defines levels of contract for each function of the service. These levels involve the syntax, semantics, synchronization and QoS of the function. The learning technique related attributes of the Function class are represented using the Learning class. Thus, the service instance of a XML MSAS is an intermediate representation of the service. This intermediate form decouples the GUI and the translator module. This allows seamlessly attaching translators for different languages to the GUI module.

5.1.3. MS_Translator
The MS_Translator is responsible for translating the intermediate service representation into corresponding Java stubs. This module implements the mapping of the multiple levels in the MSAS onto Java constructs as described in chapter 4.
Figure 5.4: Class Diagram for intermediate service representation
5.2. Case Studies

This section describes an empirical validation of the MSAS. This empirical validation consists of case studies containing three services: the Adaptive Classifier service, the Adaptive Container Service, and the Adaptive Tracking Service for demonstrating the MSAS. These cases from different domains of Computer Science were selected to demonstrate the generic nature of the MSAS. The section also presents the stubs generated by TraMS for these services.

5.2.1. Adaptive Classifier Service

As elaborated in section 3.3.1, the adaptive classifier is an information classifier service which uses reinforcement learning technique to improve its list of Acquaintances. The MSAS (XML) for this service is presented again in Table 5.1. The Java stub generated for this MSAS (XML) is given in Table 5.2.

Table 5.1: MSAS (XML) of the Adaptive classifier service

```xml
<Service Name="Classifier">
  <Description> This service is a Information Classifier </Description>
  <InherentAttributes>
    <Author> Sucheta Phatak </Author>
    <Version>1.0 </Version>
    <DateDeployed> 2002-09-24 </DateDeployed>
    <ExecutionEnvironment> N/A </ExecutionEnvironment>
    <ComponentModel> N/A </ComponentModel>
    <Validity> 2010-06-17 </Validity>
    <Structure> N/A </Structure>
    <registrations> N/A </registrations>
  </InherentAttributes>
</Service>
```
<FunctionalAttributes>
  <TaskDescription> Classifies documents </TaskDescription>
  <AlgorithmmandComplexity> N/A </AlgorithmmandComplexity>
  <Alternatives> N/A </Alternatives>
  <Resources>
    <Architecture> N/A </Architecture>
  </Resources>
  <Designpatterns> N/A </Designpatterns>
  <Usages> Information Classification Systems </Usages>
  <Aliases> InformationClassifier </Aliases>
</FunctionalAttributes>

<FunctionsAndContract>
  <Function Name="Classify">
    <Syntax>
      <Adapts> Sort; Assort </Adapts>
      <Return Name="result" Type="DocumentClass">
        <Adapts>String</Adapts>
      </Return>
      <Arguments>
        <Adapts order="true"/>
        <Argument Name="doc" Type="Document">
          <Adapts>URL</Adapts>
        </Argument>
        <Argument Name="thesaurus" Type="String" Default="NetworkThesaurus.txt"/>
        <Argument Name="return_Type" Type="int" Default="0"/>
        <Argument Name="postProc" Type="int" Default="0"/>
      </Arguments>
    </Syntax>
  </Function>
</FunctionsAndContract>
<Exceptions>RemoteException;FileNotFoundException; UnadaptiveException</Exceptions>
</Syntax>

<Semantics>
<Predicate Expression="doc!=null &&
doc.getClass().toString().equals("Document") &&
doc.getType().equals("Text")"></Predicate>
<Adapts>doc!=null &&
doc.getClass().toString().equals("Document") &&
(doc.getType().equals("Text") ||
(doc.getType().equals("MS Word Doc"))</Adapts>
</Predicate>
<Predicate Expression="result!= null">
<Adapts>result!= null &&
!result.getDocumentClass.IsEmpty()</Adapts>
</Predicate>
<Invariant> this.ACQ ! = Null </Invariant>
</Semantics>

<Synchronization type="Multi-Threaded">
<Policy Name="Mutual Exclusion">
<Adapts>FCFS </Adapts>
</Policy>
<Implementation Technique="Semaphore">
<Adapts>Monitors</Adapts>
</Implementation>
</Synchronization>

<QoS>
<Parameter Name="Availability" Unit="percentage">
   <Value From="100" To="100" />
</Parameter>

<Parameter Name="Turn-around-time" Unit="ms">
   <Value From="3" To="130" />
</Parameter>

<Parameter Name="Accuracy" Unit="percentage">
   <Value From="10" To="100" />
   <Indicator Current="30" />
</Parameter>

<Learning>
   <LearningType>Reinforcement</LearningType>
   <Feedback>Reward</Feedback>
   <LearningLatency Value="10" Unit="number of interactions" />
</Learning>

</QoS>

</Function>

<Function Name="Classify_Feedback">

<Syntax>
   <Return Type="void" />
   <Arguments>
      <Argument Name="Reward" Type="int" />
   </Arguments>
</Syntax>

<Semantics>
   <PreCondition
   Expression="this.prev_call.equals("Classify")" />
   <PostCondition
   Expression="this.reward.registered==true" />
</Semantics>

</Function>
<Invariant> this.newClassifyCall == false</Invariant>
</Semantics>
<Synchronization type="Single-Threaded"/>
<QoS>
  <Parameter Name="Availability" Unit="percentage">
    <Value From="100" To="100"/>
  </Parameter>
  <Parameter Name="Turn-around-time" Unit="ms">
    <Value From="2" To="2"/>
  </Parameter>
</QoS>
</Function>

<Function Name="Classify_init">
<Syntax>
  <Return Type="void"/>
  <Arguments>
    <Argument Name="policy" Type="int"/>
  </Arguments>
  <Exceptions> UnadaptiveException </Exceptions>
</Syntax>
<Semantics>
  <PreCondition Expression="policy < 2 && policy > -1"/>
  <PostCondition Expression="this.sync_policy != null"/>
  <Invariant> true</Invariant>
</Semantics>
<Synchronization type="Multi-Threaded">
  <Policy Name="ReaderWriter"/>
</Synchronization>
<QoS>
As seen from Table 5.1, the adaptive classifier service provides adaptations at multiple levels of the specification. Adaptations such as the argument type, the return type, the adaptive pre-conditions, and the adaptive post-conditions increase the usability of the service. Clients can provide the URL of the document or they can get a Microsoft Word Document classified. Although, the usability is increased, the adaptations affect the turn-around-time of the classify function, which is evident from the QoS level specification. The QoS level specification also indicates the usage of learning technique by the classify function. The syntax level of classify function indicates that it throws “UnadaptableException”. This exception is thrown if some adaptive behavior, which is not supported by the service, is expected by the clients. Thus, the multi-level specification gives a detailed insight of the adaptive classifier service.

Table 5.2: Stub generated by TraMS for Adaptive Classifier Service

```java
public class Classifier {
    public Object Classify(String[] args) throws RemoteException, FileNotFoundException {
    }
```
float Classify_Turn-around-time = System.currentTimeMillis();

Document doc;
Object retrun_result = null;
String thesaurus = "defaultThes.txt";
int return_Type = 0;
int postproc = 0;
//Get values of arguments from key-value pairs
for(int i=0; i< args.length; i++)
{
    String NTV[] = args[i].split(";");
    if(NTV[1].equals("doc"))
    {
        if(NTV[0].equals("URL"))
        {
            URL doc_a = new URL(NTV[2]);
            /*TODO Code to convert doc_a to doc*/
        }
    }
    else if(NTV[1].equals("thesaurus"))
    {
        thesaurus = NTV[2];
    }
    else if(NTV[1].equals("return_Type"))
    {
        return_Type = Integer.parseInt(NTV[2]);
    }
    else if(NTV[1].equals("postproc"))
    {
postproc = Integer.parseInt(NTV[2]);
}
}

if(doc!=null &&
doc.getClass().toString().equals("Document") &&
(doc.getType().equals("Text") || (doc.getType().equals("Word Doc")) ))
{
    /*TODO Code to transform to base pre-
    condition : doc!=null && doc.getClass().toString().equals("Document") && doc.getType().equals("Text") */
}

if(this.synch_policy==0)
{
    Semaphore sem = new Semaphore(1);
    sem.acquire();
    DocumentClass result = Classify_Core(doc, thesaurus, return_Type, postproc);
    sem.release();
}
else if(this.synch_policy==1)
{
    DocumentClass result = Classify_Synchronized(doc, thesaurus, return_Type, postproc);
}
if(this.synch_policy==2)
{
    ReentrantLock lock = new
ReentrantLock(true);
    lock.lock();
    try {
        DocumentClass result = Classify_Core(doc, thesaurus, return_Type, postproc);
    }
    finally {
        lock.unlock();
    }
    }
    if(postProc == 1) {
    /* TODO Code to transform from base post-condition : result != null To: result != null && !result.getDocumentClass.IsEmpty()*/
    assert (result != null && !result.getDocumentClass.IsEmpty()) : "Adaptive PostCondition Violated";
    }
    if(return_Type==1) {
    /*TODO Code to Convert result to String*/
    }
    Classify_Turn-around-time = System.currentTimeMillis() - Classify_Turn-around-time;
    return return_result;
public Object Sort(String[] args) throws RemoteException, FileNotFoundException
{
    return Classify(args);
}

public Object Assort(String[] args) throws RemoteException, FileNotFoundException
{
    return Classify(args);
}

private synchronized DocumentClass Classify_Synchronized(Document doc, String thesaurus, String return_Type, int postproc) throws RemoteException, FileNotFoundException
{
    DocumentClass result = Classify_Core(doc, thesaurus, return_Type, postproc);
    return result;
}

private DocumentClass Classify_core(Document doc, String thesaurus, String return_Type, int postproc) throws RemoteException, FileNotFoundException
{
    assert (doc!=null && doc.getClass().toString().equals("Document") &&
    doc.getType().equals("Text")) :"PreCondition Violated";
    assert (this.ACQ != Null) :"Invariant
public void Classify_Feedback(int reward) {
    assert (this.prev_call.equals("Classify")):
    “PreCondition for Classify_Feedback violated”;
    assert (this.newClassifyCall == false):
    “Invariant for Classify_Feedback violated”;

    if (reward > 0) {
        this.Classify_availability =
        ((this.Classify_numReq - 1) * this.Classify_availability + 100) / this.Classify_numReq;
    }
    else {
        this.Classify_availability =
        ((this.Classify_numReq - 1) * this.Classify_availability - 100) / this.Classify_numReq;
    }
    /* code to register the feedback */
    assert (this.newClassifyCall == false):
    “Invariant for Classify_Feedback violated”;
    assert (this.reward.registered == true):
    “PostCondition for Classify_Feedback violated”;
public void Classify_init(int policy) throws UnadaptiveException
{
    assert(policy < 2 && policy > -1): “PreCondition for Classify_init violated”;

    if(!this.syncPolicylock.writeLock().tryLock())
        throw new UnadaptiveException();
    this.synch_policy = policy;
    this.syncPolicylock.writeLock().unlock();
    assert(this.synch_policy!=null): “PostCondition for Classify_init violated”;
}

This Table 5.2 represents the translation of the MSAS XML of an adaptive classifier to Java stubs by the TraMS. As seen from this table, these stubs contain the necessary constructs for handling different types of adaptations. Thus, the stubs generated for adaptive classifier using the TraMS assist the service developer in implementing the service.

5.2.2. Adaptive Container Service
This service provides a generic container, allowing clients to insert and delete elements from it. It implements a bounded buffer synchronization policy for achieving concurrency control. This service is typically used in systems implementing stack or queues. The MSAS (XML) for this service is presented in the Appendix (Table A.1). The Java stub generated for this MSAS is presented in
the Appendix (Table A.2). The adaptive container provides adaptive functions such as, create, insert, and remove. As indicated in the MSAS, the insert and remove functions can act as push and pop function of a stack or enqueue and dequeue function of a queue respectively. The required skeleton code for handling these adaptations is presented in the Java stubs generated by the TraMS. This stub indicates the code generated for handling adaptive post-conditions. This code ensures the adaptive post-conditions at the end of the function execution.

5.2.3. Adaptive Tracking Service
This service is used by Distributed Tracking Systems [JOSHI08] to track an Object in an environment given its “Marker”. It provides the functionality for getting the real-time coordinates of Objects. Table A.3, in the Appendix, presents the MSAS (XML) for this service and its corresponding Java stub is presented in Table A.4. The MSAS for adaptive tracking service indicates the return type and argument type adaptation. It also indicates that, the tracking function handles the case where a marker pattern, which is not registered, is sent for tracking. The adaptive post-condition for the tracking service indicates the transformation of coordinates with respect to the origin provided by the clients. Corresponding stub code, indicated in Table A.4, presents the skeleton code for handling all these adaptations. Service developers can use this skeleton code to implement the actual adaptive tracking service.

5.3. Empirical Evaluation for Adaptive Classifier
This section elaborates on the implementation of the Adaptive Classifier service and compares it with a plain, i.e., non-adaptive, classifier service. The parameters used for the comparison are the disk footprint, Lines Of Code, the turn-around-time and accuracy.
5.3.1. Plain Classifier Service
A plain classifier service is a document classifier providing no adaptations. It implements the document classification algorithm as discussed in [RAJ97]. This service collaborates with similar plain classifier services to achieve a classification of given documents. These collaborators are called “Acquaintances” of the classifier service. In this experimental setup, each classifier has three acquaintances, which are indicated using a configuration file. The classifier service randomly chooses one of the three acquaintances for delegating the classification request if it fails to classify the requested document. Every classifier refers to a thesaurus for classifying a document. A thesaurus contains set of words which are searched for in the document to be classified. The Multi-level specification of such a plain classifier is given in Table 3.2. This service is implemented as a Java RMI service for experimental evaluation.

5.3.2. Adaptive Classifier
The Adaptive Classifier service provides the similar functionality as the plain classifier along with run-time adaptations. This service has a dynamic list of acquaintances chosen from available classifiers. This list is updated using the feedback of classification given by the clients. Adaptive classifier also uses a Thesaurus for classification and gives the flexibility to the clients to specify the thesaurus to be used for classifying their document. This service is discussed in detail in section 5.2.1. For experimental evaluation, the service is implemented as a Java RMI service.

5.3.3. Empirical Evaluation
The experimental study involves two different setups. First consisting of the adaptive classifiers and second consisting of the plain classifiers. Each setup comprises of ten distributed classifier services. Each classifier service has three acquaintances which are themselves distributed classifiers. Clients are
developed as Java RMI clients. Following sections provide, in detail, an analysis of the results for these two setups. These sections discuss the parameters such as disk footprint, Lines Of Code (LOC), turn-around-time and accuracy. These parameters are directly influenced by different adaptations and hence, they are crucial in assessing the cost and benefits of adaptation.

5.3.3.1. Disk Footprint and LOC
Following table indicates the results for both the services on a Windows Vista operating system:

<table>
<thead>
<tr>
<th>Service</th>
<th>Disk Size</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptive Classifier</td>
<td>24KB</td>
<td>781</td>
</tr>
<tr>
<td>Plain Classifier</td>
<td>7 KB</td>
<td>209</td>
</tr>
</tbody>
</table>

As indicated in Table 5.3, the adaptive service implementation of the classifier service has almost 200% increase in the LOC. The increase in LOC is due to the stub generated by the TraMS and the code inserted by the service developer to actually handle the adaptations. Out of this, half of the increase accounts to the code added for implementing the reinforcement learning algorithm. The increase in LOC is directly proportional to the growth in disk size. There is a 200% increase in disk size for the adaptive classifier service.

5.3.3.2. Turn-Around-Time
The section presents an experimental analysis of the turn-around-time for both these services. The first experimental setup consisted of 10 documents sent for classification to the plain classifier. All these documents are from the networking domain. This experiment evaluates the average time taken by a non-adaptive
classifier to classify a document. Following Table 5.4 indicates the average time taken for classifying these 10 documents. The non-adaptive classifier to which clients are making the requests has the thesaurus from the domain of computer languages. The three acquaintances of this classifier are from the Network, Operating System and Web domains. The non-adaptive classifier delegates the classification request to a randomly chosen acquaintance for every request from the client.

The Table 5.4 compares the behavior of the non-adaptive classifier and the adaptive classifier with no adaptation. The requests sent to the adaptive classifier are identical to the 10 documents sent to the Plain classifier. For each of these 10 documents, 10 requests are sent, and the average turn-around-time is considered.

Table 5.4: Turn Around time required for classifying 10 different documents by Non-adaptive classifier and Adaptive classifier with no adaptations

<table>
<thead>
<tr>
<th>Document</th>
<th>Non-adaptive Classifier, Turn-around-time (ms)</th>
<th>Adaptive Classifier with no Adaptation, Turn-around-time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.1</td>
<td>15.3</td>
</tr>
<tr>
<td>2</td>
<td>14.6</td>
<td>14.7</td>
</tr>
<tr>
<td>3</td>
<td>14.1</td>
<td>14.1</td>
</tr>
<tr>
<td>4</td>
<td>14.3</td>
<td>14.3</td>
</tr>
<tr>
<td>5</td>
<td>12.7</td>
<td>12.7</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>14.2</td>
<td>14.3</td>
</tr>
<tr>
<td>8</td>
<td>12.6</td>
<td>12.7</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>11.2</td>
</tr>
<tr>
<td>10</td>
<td>12.6</td>
<td>12.7</td>
</tr>
</tbody>
</table>
The non-adaptive classifier randomly selects the acquaintance for every request. On the other hand, selection of acquaintance for the adaptive classifier is done in the feedback function. As seen from the table, the turn-around-time of an adaptive classifier is similar to the non-adaptive classifier when no adaptations are required.

The second experimental setup consists of comparing the turn-around-time required by the adaptive classifier for different categories of adaptations, such as, the order adaptation, type adaptation, pre-condition adaptation, etc. In this setup, requests are sent to the adaptive classifier demanding different adaptations. Ten requests of the same type are sent and the turn-around-time for these requests is averaged. This setup considers requests involving the order adaptation, the argument type adaptation, the name adaptation, the number of argument adaptation, the return type adaptation, the adaptive pre-condition, the adaptive post-condition and the synchronization policy and technique adaptation. For these nine adaptations, following table indicates the average turn-around-time required by the adaptive classifier. For each of these adaptations, the document number one from the network domain is sent for classification. The Table 5.5 presents the turn-around-time for 2 functions of the adaptive classifier, the “classify” function and the “Classify_init” function. The “Classify_init” function is the function used to map synchronization adaptation as discussed in section 4.4.

<table>
<thead>
<tr>
<th>Adaptation Type</th>
<th>Classify turn-around-time (ms)</th>
<th>Classify_init turn-around-time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order Adaptation</td>
<td>15.3</td>
<td>0</td>
</tr>
<tr>
<td>Name Adaptation</td>
<td>15.3</td>
<td>0</td>
</tr>
<tr>
<td>Argument Type Adaptation</td>
<td>125.5</td>
<td>0</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------</td>
<td>---</td>
</tr>
<tr>
<td>Number of Arguments</td>
<td>15.4</td>
<td>0</td>
</tr>
<tr>
<td>Return Type Adaptation</td>
<td>15.4</td>
<td>0</td>
</tr>
<tr>
<td>Pre-Condition Adaptation</td>
<td>15.7</td>
<td>0</td>
</tr>
<tr>
<td>Post-Condition Adaptation</td>
<td>15.3</td>
<td>0</td>
</tr>
<tr>
<td>Synchronization Technique Adaptation</td>
<td>15.5</td>
<td>2</td>
</tr>
<tr>
<td>Synchronization Policy Adaptation</td>
<td>15.4</td>
<td>2</td>
</tr>
</tbody>
</table>

As seen from Table 5.4, the average time required to classify the document number one by the adaptive service with no adaptation is 15.3ms. Time required to classify the same document with the return type adaptation is 15.4ms. Thus, the return type adaptation adds an overhead of 0.1 ms to the turn-around-time. Similarly, the name adaptation, the number of argument adaptation, the return type adaptation, the pre-condition and post-condition adaptations, the synchronization technique and policy adaptations add a small overhead on the turn-around-time of the classify function. As shown in the table, the synchronization technique and policy adaptation has an extra overhead in terms of processing the Classify_init function, which has a turn-around-time of 2ms on average. The most expensive adaptation for the classifier service is the argument type adaptation, where a URL is accepted rather than a document. This adaptation requires downloading the contents of the URL from a remote machine and storing it locally in form of a document. This document is then classified by the classifier, thus, leading to a substantial overhead.

Along with all these above adaptations, the adaptive classifier service uses the reinforcement learning technique to dynamically adapt the acquaintances. This adaptation does not affect the turn-around-time of the classify function. This is because all the learning code is incorporated in the feedback function. This
feedback function has a turn-around-time of 2ms on average. This counts towards the overhead of learning technique used by adaptive service.

The next experimental setup considers classifying the document number 1, incorporating multiple adaptations at a time.

Table 5.6: Turn-around-time for adaptive service with multiple adaptations

<table>
<thead>
<tr>
<th>Adaptations</th>
<th>Turn-around-time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name + order + return Type + pre-condition + post-condition +</td>
<td>15.10</td>
</tr>
<tr>
<td>synchronization policy</td>
<td></td>
</tr>
<tr>
<td>Name + Number of arguments + return Type + pre-condition +</td>
<td>15.10</td>
</tr>
<tr>
<td>post-condition + synchronization technique</td>
<td></td>
</tr>
<tr>
<td>Order + return Type + post-condition + argument Type</td>
<td>126.3</td>
</tr>
<tr>
<td>number of arguments + post-condition + argument Type +</td>
<td>126.6</td>
</tr>
<tr>
<td>synchronization policy</td>
<td></td>
</tr>
</tbody>
</table>

As seen from the Table 5.6, the adaptive classifier performing multiple adaptations for one request has the turn-around-time almost equal to addition of average turn-around-time of adaptive classifier with no adaptation and overhead due to individual adaptations. This is clearly evident from the last two rows of the table 5.6, which indicates multiple adaptations including the argument type adaptation.

Similar experiments with multiple adaptations were carried out for other 9 documents used in Table 5.4. These results confirmed that the overhead on turn-around-time for the adaptive classifier with multiple adaptations is almost equal to the addition of overheads of individual adaptations.
Conclusively, this experimental analysis gives following insights on effects of adaptation on turn-around-time.

- Thus, the effect of multiple adaptations cumulates the effects of individual adaptations.

5.3.3.3. **Accuracy**

The accuracy of classification is measured in terms of the success of classification. Figures 5.5 and 5.6 indicate the variation in success of classification for 100 classification requests for the plain classifier and the adaptive classifier respectively. The graphs indicate the number of successful classifications for a group of classification requests. The 100 client requests are grouped in 10 groups of 10 requests each. The 100 requests in both the adaptive and non-adaptive setups are identical with same order. All the 100 requests are for one of the 10 documents of Network domain. As the plain classifier has the thesaurus of Language domain, local classification fails. The three acquaintances of this classifier are of Network, Operating System and Web domain. Whenever, the acquaintance with Network domain is chosen for delegation of the request, classification succeeds. Thus, amongst the first 10 classification requests sent to the plain classifier, 5 classifications were sent to this acquaintance.
The adaptive setup consists of identical 100 requests to the Adaptive classifier. The adaptive classifier has the default thesaurus of Language domain, but it provides the adaptation for specifying thesaurus as part of the classification request. Hence, the adaptive classifier using the randomly sent thesaurus can classify the document locally. This increases the classification success rate. Apart from this the important adaptation which affects the classification success is learning technique implemented by the adaptive classifier.

Figure 5.5: Classification success variations for Plain Service
Figure 5.6 Classification success variations for Adaptive Service

Figure 5.5 indicates random variation in successful classifications for the plain classifier. On an average, 5 classification requests are successful amongst 10 classification requests. On the other hand, for an adaptive classifier, the classification success rate increases as the number of requests increase. For the initial phase of classification request, the success rate is low and it increases as the classifier learns from its experience. The reinforcement learning algorithm used by the adaptive classifier is such that it explores different acquaintances rather than always choosing the best acquaintance [MUKHO 03]. Hence, the classification success rate increases slowly as the algorithm evolves. Thus, the average classification success for adaptive classifier is 7 amongst 10 requests. This shows that the average increase in successful classification by the adaptive classifier is 40% and the maximum increase is 100%. The number of iterations required for the adaptive classifier to achieve 100% classification is the learning latency of classifier. The learning latency for adaptive classifier is on average 110 requests.
Considering the statistics given in this section following conclusions can be deduced:

- Implementing adaptive services have an overhead in the form of an increase in the code size than their non-adaptive counterparts. This increase depends on the number of adaptations, types of adaptation and the implementation mechanism of the adaptation provided by the adaptive service. For the adaptive classifier service, this increase is almost 200%.

- The effect of adaptation is clearly evident from the analysis of the turn-around-time for the classify function. In general, the turn-around-time for an adaptive service will depend on the types of adaptations and techniques of adaptation performed at run-time. Increase in turn-around-time for adaptations counts towards the cost of adaptation.

- The adaptive behavior of the classifier service also affects the success of classification. Two important adaptations affect this parameter, adaptation for the default argument of thesaurus provided by the clients and the adaptation using reinforcement learning technique. Both these adaptations increase the classification success rate. This improvement is at the cost of additional turn-around-time of 2 ms for the feedback function, increase in disk footprint and LOC.

- Thus, the adaptive classifier service provides better accuracy of results in terms of classification success rate than the non-adaptive counterpart. This improvement is at the cost of an increase in the code size and an increase in the turn-around-time.

Summarizing, this chapter discusses the implementation details of the TraMS and validates its usage by considering three adaptive services from different domains of Computer Science. This chapter also empirically evaluates the effects of different types of adaptation on the adaptive classifier service. This experimental evaluation gives the insight of cost of adaptation and benefits of adaptation.
CHAPTER 6. CONCLUSION AND FUTURE WORK

As seen in the previous chapters, this thesis provides a detailed design and implementation of a multi-level specification for adaptive services. The work done by this thesis provides the necessary constructs and tools for facilitating adaptive service development. The thesis starts with discussing the motivation behind the design of the MSAS. The work done by the research community in this area is then presented. This is followed by the detailed description of the design of different constructs in MSAS and their mapping with Java. Implementation of the TraMS for the MSAS is then discussed followed by case studies for validating this work. Conclusively, this chapter summarizes the work done by the thesis and presents future extensions for this work.

6.1. Conclusions

Based on the research work performed by this thesis, following conclusions can be drawn.

1. The Multi-level Specification for Adaptive Services (MSAS) presented in this thesis is application domain independent and comprehensive. This specification serves as blue-print of the service for the developers and as an adaptive contract for the clients of the service. The specification provides constructs for representing adaptations at different levels, i.e., syntax, semantics, synchronization and QoS levels, of contract.

2. The thesis provides a detailed mapping for the constructs provided by the MSAS to the Java language. This mapping is realized by implementing a translator for converting MSAS to Java stubs.
3. The effectiveness of the MSAS is empirically validated using three case study services from different domains of computer science.

4. Finally, the cost of adaptation is empirically evaluated for the Adaptive Classifier Service. The analysis shows that adaptive services have the benefit of increasing the usability by adapting to the change in requirements. Adaptive services also optimize accuracy, or other QoS parameters. Although, along with these benefits, adaptive services suffer from substantial increase in the Lines Of Code, and the storage size.

6.2. Contributions
The important contributions of the thesis are as follows:

1. The implementation and design of Multi-level Specification for Adaptive Services which is comprehensive and generic. This involved investigation and analysis of: a) multi-level specification, b) different adaptation types, c) adaptations at different levels of specification.

2. A detailed mapping of the Multi-level specification to Java language and its validation by implementing a Translator based on it. This involved indentifying the transformation for each of the constructs in MSAS to Java. This translator creates skeleton code which representing the adaptive service.

3. Validation and verification of the translator using case study services.

4. Empirical evaluation of cost and benefits of adaptation for the Classifier service. This evaluation is derived using experimental analysis which assists in identifying merits and demerits of adaptation.
6.3. **Future Work**

Several future extensions to this research work are possible and a few of which are suggested below:

1. Adaptation at the temporal level of contract suggested by [TIL07] can be considered and the necessary constructs can be added to the MSAS. A mapping for these new constructs can also be defined and implemented.

2. The adaptive Code generation framework can be implemented to generate stub code for adaptive services at run-time. This framework will generate the code to handle adaptive behavior at run-time using a knowledge base.

3. The mapping suggested by the thesis can be validated using other object oriented languages such as C++ and C#.

4. An aspect oriented framework can be used to implement adaptations efficiently by weaving the adaptation aspect to the core functionality at run-time.

5. A framework for estimating cost of adaptation can be implemented. This framework, given a Multi-level specification for an Adaptive service, will generate a detailed estimation of cost of adaptations indicated by the specification.

6.4. **Summary**

The research work presented this thesis provides a generic and comprehensive Multi-level Specification for Adaptive Services. Formal constructs are suggested for representing different types of adaptations at each level of contract. A detailed mapping is defined by the thesis for transforming the MSAS into Java stubs. The specification helps service developers to create a blue-print of their adaptive service. The TraMS provided by the thesis can be used by service developers to convert this specification to Java stubs to which they can add their implementation code. The specification and translator is validated using three case studies. A detailed analysis of empirical evaluation of cost and benefits of adaptation is
provided for the classifier service. Conclusively, the work done by the thesis will assist the adaptive service development by providing required specification constructs and associated tools.
LIST OF REFERENCES


[VAM06] Vamshi Cheekati, “Translation of the UMM specification to a Language Specific Implementation”, M.S. Project, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, April 2006.


Table A.1: MSAS (XML) for Adaptive Container Service.

```xml
<Service Name="OrderedContainer"/>
<Description> This service is a OrderedContainer</Description>

<InherentAttributes>
<Author> Sucheta Phatak </Author>
-Version>1.0 </Version>
<DateDeployed> 2002-09-24 </DateDeployed>
<ExecutionEnvironment> None </ExecutionEnvironment>
<ComponentModel> None </ComponentModel>
<Validity> 2010-06-17 </Validity>
<Structure> None </Structure>
<registrations> None </registrations>
</InherentAttributes>

<FunctionalAttributes>
<TaskDescription> Acts as an Ordered Container </TaskDescription>
<AlgorithmandComplexity>O(n);O(nlogn)</AlgorithmandComplexity>
<Alternatives>Stack;Queue</Alternatives>
<Resources>
  <Architecture> N/A </Architecture>
</Resources>
```
<Designpatterns>Container</Designpatterns>
<Usages>BufferManagement</Usages>
<Aliases>Buffer</Aliases>
</FunctionalAttributes>

<FunctionsAndContract>
<Function Name="Create">
<Syntax>
   <Adapts>Init;GetContainer</Adapts>
   <Return Name="container" Type="OrderedContainer"/>
   <Arguments>
      <Argument Name="size" Type="int"/>
      <Argument Name="postproc" Type="int"/>
   </Arguments>
   <Exceptions> UnAdaptableException </Exceptions>
</Syntax>

<Semantics>
   <PreCondition Expression="true"/>
   <PostCondition Expression="container.NumberOfElements>=0">
      <Adapts>container.NumberOfElements>0</Adapts>
   </PostCondition>
   <Invariant>true</Invariant>
</Semantics>

<Synchronization Type="Multi-Threaded">
   <Policy Name="Bounded Buffer"/>
   <Implementation Technique="Semaphore"/>
</Synchronization>
<QoS>
   <Parameter Name="Turn-Around-Time" Unit="ms">
      <Value From="3" To="5"/>
   </Parameter>
</QoS>
</Function>

<Function Name="Insert">
   <Syntax>
      <Adapts> Push;Enqueue </Adapts>
      <Return Name="container" Type="OrderedContainer"/>

      <Arguments>
         <Adapts order="true"/>
         <Argument Name="element" Type="T"/>
         <Argument Name="postproc" Type="int"/>
      </Arguments>

      <Exceptions>UnAdaptableException;ContainerFullException </Exceptions>
   </Syntax>

   <Semantics>
      <PreCondition Expression="!this.IsFull()"/>
      <PostCondition Expression="this.prevSize ==
this.NumberOfElements -1">
         <Adapts>this.prevSize == this.NumberOfElements -1 &&
         this.lastElement==element ; this.prevSize ==
         this.NumberOfElements -1 && this.sorted == true;
         this.prevSize == this.NumberOfElements -1 &&
         this.firstElement==element</Adapts>
      </PostCondition>
   </Semantics>
</Function>
<Invariant>true</Invariant>
</Semantics>

<Synchronization Type="Multi-Threaded">
  <Policy Name="Bounded Buffer"/>
  <Implementation Technique="Semaphore"/>
</Synchronization>

<QoS>
  <Parameter Name="Turn-Around-Time" Unit="ms">
    <Value From="3" To="5"/>
  </Parameter>
</QoS>

<Function Name="Remove">
  <Syntax>
    <Adapts> Pop;Dequeue </Adapts>
    <Return Name="element" Type="T"/>
    <Arguments>
      <Argument Name="postproc" Type="int"/>
    </Arguments>
    <Exceptions>
      UnAdaptableException;ContainerEmptyException
    </Exceptions>
  </Syntax>
  <Semantics>
    <PreCondition Expression="!this.IsEmpty()"/>
    <PostCondition Expression="this.prevSize ==
      this.NumberOfElements +1">
      <Adapts>this.prevSize == this.NumberOfElements +1 &
    </Adapts>
  </Semantics>
</Function>
the prevlastElement==element ; this.prevSize ==
this.NumberOfElements +1 && this.sorted == true;
this.prevSize == this.NumberOfElements +1 &&
this.prevfirstElement==element </Adapts>
</PostCondition>
</Invariant>true</Invariant>
</Semantics>

<Synchronization Type="Multi-Threaded">
   <Policy Name="Bounded Buffer"/>
   <Implementation Technique="Semaphore"/>
</Synchronization>

<QoS>
   <Parameter Name="Turn-Around-Time" Unit="ms">
      <Value From="3" To="5"/>
   </Parameter>
</QoS>

</Function>
</FunctionsAndContract>
</Service>

Table A.2: Java stub generated for MS(XML) of Adaptive Container Service.

```
public class OrderedContainer {
    public OrderedContainer Create(int size, int postproc)
        throws UnAdaptableException {
        OrderedContainer container = Create_Core(size,
```
postproc);  
  if(postProc == 1)  
  {  
    /*Insert Code to transform from base post-
condition :container.NumberOfElements>=0 To:
container.NumberOfElements>0*/  
    assert (container.NumberOfElements>0) :  
    "Adaptive PostCondition Violated";  
  }  
}  
}  

public OrderedContainer Init(int size,int postproc) throws UnAdaptableException  
{  
    return Create(size, postproc);  
}  
public OrderedContainer GetContainer(int size,int postproc) throws UnAdaptableException  
{  
    return Create(size, postproc);  
}  
public OrderedContainer Create(int size) throws UnAdaptableException  
{  
    int postproc = 0;  
    return Create(size, postproc);  
}  
private OrderedContainer Create_core(int size,int postproc) throws UnAdaptableException  
{  
    //INSERT Core functionality
assert (container.NumberOfElements>=0) ;
:"PostCondition Violated";

public OrderContainer Insert(String[] args) throws UnAdaptableException
{
    //Declaring arguments
    T element;
    int postproc = 0;
    //Get value of arguments from key-value pairs
    for(int i=0; i< args.length; i++)
    {
        String NTV[] = args[i].split("," );
        if(NTV[1].equals("element"))
        {
            element = new T(NTV[2]);
        }
        else if(NTV[1].equals("postproc"))
        {
            postproc = Integer.parseInt(NTV[2]);
        }
    }
    
    OrderContainer container = Insert_Core(element, postproc);

    if(postProc == 1)
    {
        /*INSERT Code to transform from base post-
        condition :this.prevSize == this.NumberOfElements -1 To:
this.prevSize == this.NumberOfElements -1 &&
this.lastElement==element */
    assert (this.prevSize ==
this.NumberOfElements -1 && this.lastElement==element ) :
"Adaptive PostCondition Violated";
}
else if(postProc == 2)
{
    /*INSERT Code to transform from base post-
    condition :this.prevSize == this.NumberOfElements -1 To:
    this.prevSize == this.NumberOfElements -1 && this.sorted ==
    true*/
    assert ( this.prevSize ==
this.NumberOfElements -1 && this.sorted == true) :
"Adaptive PostCondition Violated";
}
else if(postProc == 3)
{
    /*INSERT Code to transform from base post-
    condition :this.prevSize == this.NumberOfElements -1 To:
    this.prevSize == this.NumberOfElements -1 &&
    this.firstElement==element*/
    assert ( this.prevSize ==
this.NumberOfElements -1 && this.firstElement==element) :
"Adaptive PostCondition Violated";
}
}

public OrderContainer Push(String[] args) throws
UnAdaptableException
{
    return Insert(args);
public OrderContainer Enqueue (String[] args) throws UnAdaptableException
{
    return Insert(args);
}
private OrderContainer Insert_core(T element, int postproc) throws UnAdaptableException
{
    assert (!this.IsFull()) : "PreCondition Violated";
    /*INSERT Core functionality*/
    assert (this.prevSize == this.NumberOfElements - 1) : "PostCondition Violated";
}
public T Remove(int postproc) throws UnAdaptableException
{
    T element = Remove_Core(postproc);
    if(postProc == 1)
    {
        /* INSERT Code to transform from base post-condition :this.prevSize == this.NumberOfElements +1 To:
            this.prevSize == this.NumberOfElements +1 &&
            this.prevlastElement==element */
        assert (this.prevSize ==
            this.NumberOfElements +1 && this.prevlastElement==element ) : "Adaptive PostCondition Violated";
    }
    else if(postProc == 2)
    {
        /* INSERT Code to transform from base post-
condition : this.prevSize == this.NumberOfElements + 1 To:
this.prevSize == this.NumberOfElements + 1 && this.sorted ==
true*/
    assert (this.prevSize ==
this.NumberOfElements +1 && this.sorted == true): "Adaptive PostCondition Violated";
}
else if(postProc == 3)
{
    /* INSERT Code to transform from base post-
condition : this.prevSize == this.NumberOfElements + 1 To:
this.prevSize == this.NumberOfElements +1 &&
this.prevfirstElement==element*/
    assert (this.prevSize ==
this.NumberOfElements +1 && this.prevfirstElement==element): "Adaptive PostCondition Violated";
}
public T Pop(int postproc) throws
UnAdaptableException
{
    return Remove(postproc);
}
public T Dequeue (int postproc) throws
UnAdaptableException
{
    return Remove(postproc);
}
public T Remove() throws UnAdaptableException
{
    int postproc = 0;
return Remove(postproc);
}
private T Remove_core(int postproc) throws UnAdaptableException {
    assert (!this.IsEmpty()) : "PreCondition Violated";
    /*INSERT Core functionality*/
    assert (this.prevSize == this.NumberOfElements +1) : "PostCondition Violated";
}

Table A.3: MSAS (XML) for Adaptive Tracking Service

<Service Name="Tracker">
<Description> This service is a Tracker service</Description>

<InherentAttributes>
<Author> Sucheta Phatak </Author>
<Version>1.0 </Version>
<DateDeployed> 2002-09-24 </DateDeployed>
<ExecutionEnvironment> None </ExecutionEnvironment>
<ComponentModel> None </ComponentModel>
<Validity> 2010-06-17 </Validity>
<Structure> None </Structure>
<registrations> rmi://pegasus.cs.iupui.edu </registrations>
</InherentAttributes>
<FunctionalAttributes>
<TaskDescription> Tracks an Object with given pattern </TaskDescription>
<AlgorithmandComplexity> None </AlgorithmandComplexity>
<Alternatives>N/A</Alternatives>
<Resources>
  <Architecture> Optical Trackers </Architecture>
</Resources>
<Designpatterns> None </Designpatterns>
<Usages> Augmented Reality; Distributed Tracking System </Usages>
<Aliases> LocatorService </Aliases>
</FunctionalAttributes>
<FunctionsAndContract>
<Expression Name="Track"
  Syntax>
    <Adapts> Find; Search; GetLocation; Locate </Adapts>
    <Return Name="location" Type="Coordinates">
      <Adapts> float[] </Adapts>
    </Return>
    <Arguments>
      <Argument Name="pattern" Type="Pattern"/>
      <Argument Name="Origin" Type="Coordinates" Default="0,0,0">
        <Adapts> float[] </Adapts>
      </Argument>
      <Argument Name="return_Type" Type="int" Default="0"/>
    </Arguments>
    <Exceptions> UnAdaptableException; UntrackableException </Exceptions>
</Expression>
/

</_exceptions>
</syntax>

<semantics>

<precondition expression="pattern!=null &&
pattern.registered==true">

<adapts>pattern!=null</adapts>
</precondition>

<postcondition
expression="location.Origin.toString()==Origin"/>

<invariant>NumberOfCameras>1</invariant>
</semantics>

<synchronization type="single-threaded"/>

<qos>

<parameter name="turn-around-time" unit="ms">

<value from="30" to="50"/>
</parameter>

<parameter name="accuracy" unit="cm">

<value from="4" to="6"/>
</parameter>

</qos>

</function>
</functionsandcontract>
</service>

---

Table A.4: Java stub generated for MSAS (XML) of Adaptive Tracking Service

| public class Tracker |
public Object Track(Pattern pattern, Object Origin, int return_Type) throws UnAdaptableException, UntrackableException
{
    if (Origin.getClass().getName().equals("float[]"))
    {
        //Code to Convert float[] to Coordinates
    }
    if (pattern != null)
    {
        // Code to transform to base pre-condition
        :pattern != null && pattern.registered == true
    }

    Coordinates location = Track_Core(pattern, Origin, return_Type);

    if (return_Type.equals("float[]"))
    {
        //INSERT Code to Convert location to float[]
    }
}

public Object Find(Pattern pattern, Object Origin, int return_Type) throws UnAdaptableException, UntrackableException
{
    return Track(pattern, Origin, return_Type);
}

public Object Search(Pattern pattern, Object Origin, int
public Object GetLocation(Pattern pattern, Object Origin, int return_Type) throws UnAdaptableException, UntrackableException
{
    return Track(pattern, Origin, return_Type);
}

custom Location(Pattern pattern, Object Origin, int return_Type) throws UnAdaptableException, UntrackableException
{
    return Track(pattern, Origin, return_Type);
}

custom Object Locate(Pattern pattern, Object Origin, int return_Type) throws UnAdaptableException, UntrackableException
{
    return Track(pattern, Origin, return_Type);
}

custom Object Track(Pattern pattern, Coordinates Origin) throws UnAdaptableException, UntrackableException
{
    Coordinates Origin = new Coordinates(0, 0, 0);
    return Track(pattern, Origin);
}

custom Object Track(Pattern pattern, Object Origin) throws UnAdaptableException, UntrackableException
{
    int return_Type = 0;
    return Track(pattern, Origin, return_Type);
}

private Coordinates Track_core(Pattern pattern, Coordinates Origin, int return_Type) throws
UnAdaptableException, UntrackableException

{
    assert (pattern!=null &&
            pattern.registered==true) : "PreCondition Violated";
    assert (NumberOfCameras>1) : "Invariant Violated";
    //INSERT Core functionality
    assert (NumberOfCameras>1) : "Invariant Violated";
    assert (location.Origin.toString()==Origin)
            : "PostCondition Violated";
}

Table A.5: XML Specification for MSAS

<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Service" type="serviceInfo"/>
  <xs:simpleType name="SyncPolicyType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Mutual Exclusion"/>
      <xs:enumeration value="FCFS"/>
      <xs:enumeration value="Reader Writer"/>
      <xs:enumeration value="Bounded Buffer"/>
      <xs:enumeration value="Barrier"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="serviceInfo">
    <xs:sequence>
      <xs:element name="Description" type="xs:string"/>  
    </xs:sequence>
  </xs:complexType>
</xs:schema>
<xs:element name="InherentAttributes">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Author" type="xs:string" />
      <xs:element name="Version" type="xs:string" default="1.0" />
      <xs:element name="DateDeployed" type="xs:date" minOccurs="0" />
      <xs:element name="ExecutionEnvironment" type="xs:string" minOccurs="0" />
      <xs:element name="ComponentModel" type="xs:string" minOccurs="0" />
      <xs:element name="Validity" type="xs:date" minOccurs="0" />
      <xs:element name="Structure" type="xs:string" minOccurs="0" />
      <xs:element name="Registrations" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="FunctionalAttributes" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="TaskDescription" type="xs:string" minOccurs="0" />
      <xs:element name="AlgorithmandComplexity" type="xs:string" minOccurs="0" />
      <xs:element name="Alternatives" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Resources" minOccurs="0">
  </xs:element>
  <xs:element name="Designpatterns" type="xs:string" minOccurs="0/>
  <xs:element name="Usages" type="xs:string" minOccurs="0">
    <xs:element name="Aliases" type="xs:string" minOccurs="0"/>
  </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="FunctionsAndContract">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Function" maxOccurs="unbounded">
        <xs:complexType>
          <xs:all> <!-- every element Syntax, Semantics, QoS, Synchronization must occur only once in any order -->
            <xs:element name="Syntax" minOccurs="1">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Adapts" type="xs:string" minOccurs="0"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="Return">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Adapts" type="xs:string" minOccurs="0"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:sequence>
  <xs:attribute name="Name" type="xs:string" use="optional" />  
  <xs:attribute name="Type" type="xs:string" use="required" />
</xs:complexType>
</xs:element>
<xs:element name="Arguments" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Adapts" minOccurs="0">
        <xs:complexType>
          <xs:attribute name="order" type="xs:boolean" fixed="true" use="required" />
        </xs:complexType>
      </xs:element>
      <xs:element name="Argument" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Adapts" type="xs:string" minOccurs="0"/>
          </xs:sequence>
          <xs:attribute name="Name" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Argument" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Adapts" type="xs:string" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
<xs:element name="Semantics" minOccurs="1">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="PreCondition">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="Adapts" type="xs:string" minOccurs="0" />
                        <xs:attribute name="Expression" type="xs:string" use="required" />
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="PostCondition">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="Adapts" type="xs:string" minOccurs="0" />
                        <xs:attribute name="Expression" type="xs:string" use="required" />
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="Adapts" type="xs:string" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="Expression" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
<xs:element name="Invariant" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="Synchronization" minOccurs="0" maxOccurs="1">
    <xs:complexType>
    <xs:sequence>
    <xs:element name="Policy" minOccurs="0" maxOccurs="1">
        <xs:complexType>
        <xs:sequence>
            <xs:element name="Adapts" type="SyncPolicyType" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="Name" type="SyncPolicyType"/>
    </xs:complexType>
    </xs:sequence>
    <xs:element name="Implementation" minOccurs="0" maxOccurs="1">
        <xs:complexType>
        <xs:sequence>
            <xs:element name="Adapts" type="SyncPolicyType" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="Name" type="SyncPolicyType"/>
    </xs:complexType>
    </xs:sequence>
    <xs:element name="Implementation" minOccurs="0" maxOccurs="1">
        <xs:complexType>
        <xs:sequence>
            <xs:element name="Adapts" type="SyncPolicyType" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="Name" type="SyncPolicyType"/>
    </xs:complexType>
    </xs:sequence>
</xs:element>
<xs:element name="Adapts" type="xs:string" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="Technique" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>

<xs:element name="PreCondition" type="xs:string" minOccurs="0" maxOccurs="1"/>
  <xs:element name="Action" type="xs:string" minOccurs="0" maxOccurs="1"/>
  <xs:element name="Invariant" type="xs:string" minOccurs="0" maxOccurs="1"/>
  <xs:element name="PostCondition" type="xs:string" minOccurs="0" maxOccurs="1"/>
</xs:sequence>

<xs:attribute name="Type" use="required">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Single-Threaded"/>
      <xs:enumeration value="Multi-Threaded"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
<xs:element name="QoS" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Parameter" type="QoSAttributeType" maxOccurs="unbounded"/>
      <xs:element name="Learning" minOccurs="0">
        <!-- Learning related features -->
        <xs:complexType>
          <xs:sequence>
            <xs:element name="LearningType">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="Reinforcement"/>
                  <xs:enumeration value="Supervised"/>
                  <xs:enumeration value="Unsupervised"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
            <xs:element name="Feedback" type="xs:string" default="Reward" minOccurs="0" maxOccurs="1"/>
            <xs:element name="InitialDataSet" minOccurs="0" maxOccurs="1">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="NoiseLevel">
                    <!-- ... -->
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:complexType>
  <xs:attribute name="Unit" type="xs:string" use="required" />  
  <xs:attribute name="Value" type="xs:string" use="required" />  
</xs:complexType>
</xs:element>
<xs:element name="LearningLatency">
  <xs:complexType>
    <xs:attribute name="Unit" type="xs:string" use="required" />
    <xs:attribute name="Value" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
<xs:element name="LearningLatency">
  <xs:complexType>
    <xs:attribute name="Unit" type="xs:string" use="required" />
    <xs:attribute name="Value" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:element>
</xs:all>
<xs:attribute name="Name" type="xs:string" use="required" />
</xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>
</xs:complexType>

<!--//End of Functions and Contracts -->
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="Name" type="xs:string"/> <!--Can be of type classNameType-->
</xs:complexType>

<!-- All type declarations -->

<!-- QoS Attribute type -->
<xs:complexType name="QoSAttributeType">
	<xs:sequence>
		<xs:element name="Value">
			<xs:complexType>
				<xs:attribute name="From" type="xs:string" use="required"/>
				<xs:attribute name="To" type="xs:string" use="required"/>
			</xs:complexType>
		</xs:element>
		<xs:element name="Indicator" minOccurs="0">
			<xs:complexType>
				<xs:attribute name="Current" type="xs:string" use="required"/>
			</xs:complexType>
		</xs:element>
	<!-- <xs:any minOccurs="0" maxOccurs="unbounded" namespace="##targetNamespace" -->
Figure A.1: TraMS GUI Screenshot 1
Figure A.2: TraMS GUI Screenshot 2
Figure A.3: TraMS GUI Screenshot 3
Figure A.4: TraMS GUI Screenshot 4
Figure A.6: TraMS GUI Screenshot 6