CHAPTER 4

PROPOSED ADAPTABLE MIDDLEWARE MODEL

Adaptability is emerging as a crucial enabling capability for many applications which are deployed in dynamically changing environment (Parlavantzas et al 2000). Nowadays, most of the software maintenance activities are projected on making the system adaptable to the dynamic variations in the user or environment needs. The effort expended to perform such adaptive maintenance can be reduced by introducing a dynamic adaptability feature in the system. The dynamic adaptability is defined as the ability of a software system to adapt to the behavioral or structural changes that occur either internally or externally in the operating or in the user environment while the system is running (Tarvainen 2008). Several approaches exist to achieve dynamic adaptability. They are intertwining adaptation with application behavior, application independent middleware and adaptation based middleware (Floch et al 2006, Autunes da Rocha and Endler 2006). Reflective adaptive middleware designed for using reflection mechanism is identified as a better solution for making the system adapt itself to run-time needs (Bencomo et al 2006, Cazzola et al 2004).

A reflective architecture logically models the adaptable system with two layers namely base-layer and meta-layer. The meta-layer of the reflective system is responsible for supervising the occurrence of the run-time needs to be adapted and for making the system adapt to the run-time needs. It implies that the components in the meta-layer of the reflective system are performing both the reflection and adaptation (Parlavantzas et al 2000), which reduces the
adaptability of the system. This major issue of the reflective system can be addressed through modularizing the reflective middleware that is specified as aspectizing the reflective components (Cacho et al 2006).

Another issue in the implementation of the reflective middleware using object-oriented paradigm is not providing the proper means to achieve the required level of localizing the reflection-specific concerns. Introducing aspect-oriented technology in the development of reflective based adaptive system can address the issue of localization (Bencomo et al 2008). Aspect-oriented paradigm is a new approach to software development that addresses the limitations inherent in other approaches, including object-oriented programming. Aspect-oriented approach modularizes the concerns spread over the system, which is named as cross cutting concerns (Murphy 2006). Crosscutting concerns are encapsulated in separate modules, known as aspects, so that localization can be promoted. It shows that aspect-oriented software development provides a means to modularize the crosscutting concerns in software systems (Rashid et al 2010), which results in reducing the development, maintenance and evolution costs. Many of the existing reflective middleware models have been created using traditional middleware models, which suffer from the limitations of managing the meta-level information of the components (Moreira and Araujo 2002).

The redefined dynamic aspect weaving approach proposed in this work is considered as a solution to improve the adaptability of the existing components to dynamic needs. The features of this approach are defining dynamic aspects to realize the dynamic requirements and separating the process of adapting to dynamic needs from reflection through dynamically introspecting the structure of the components and dynamically invoking the functions of the components associated with dynamic changes. The better way of realizing this approach in software is through middleware, which enhances the reusability of this solution. This proposed middleware model is
referred to as adaptable middleware for dynamic requirements. The significance of the proposed model is providing the solution exclusively for handling the dynamic changes. The middleware is defined with the components for,

- managing the activities of dynamic adaptation
- representing the dynamic requirements as dynamic aspects
- representing dynamic structural introspection
- wrapping the component service function through dynamic invocation
- weaving the dynamic aspects with the methods associated with the dynamic requirements through wrappers.

Hence, the proposed model increases the dynamic adaptability efficiency of the system without affecting the existing component structure and behavior.

The description of this middleware model includes the components that constitute this model. The implementation of the component services and the formal analysis to show the adaptability efficiency achieved using this model has been discussed in this chapter.

4.1 STRUCTURE OF THE MIDDLEWARE

The middleware depicted in Figure 4.1 has been designed using the redefined dynamic aspect weaving approach for providing the solution for dynamic adaptability. The functions of the Adaptable Middleware Model (AMM) components are: (i) Adaptation Aspect Manager (AAM) Component, which is responsible for making the system adaptable to dynamic changes in the functionalities of the component. It receives the request for dynamic
changes and uses the Change Request Parser (CRP), Aspect Generator (AG), Component Function Wrapper (CFW) and Aspect Weaver (AW) components to implement and execute the adaptable code associated with the changes; (ii) Change Request Parser Component, which parses the change request specified in xml structure and extracts the details such as name of the components and service operations associated with the changes, and details about the changes that are to be incorporated into the component services; (iii) Aspect Generator component that provides the information for creating aspect class with advice to realize the change specification; (iv) Aspect Weaver component, which binds the changes specified in the aspect with the corresponding service components through wrapper defined for the service operation using Component Function Wrapper (CFW) components;(v) Component Function Wrapper component, which dynamically invokes the component functions associated with the changes using component meta information and (vi) The Component Meta Model (CMM) extracts the information about the structure of component using introspection mechanism at run-time.

The Unified Modeling Language (UML) is used to describe the architecture of the Adaptive Middleware Model shown in Figure 4.1 as a Platform Independent Model (PIM). This PIM model can be easily transformed to any Platform Specific Model by applying model transformation rules generated using the appropriate UML tool. Each component of AMM is defined with provided and required interfaces. The Adaptation Aspect Manager component is defined with IDynChangeRequest as the provided interface and IChangeRequestParser, IAspectGenerator and IDynamicWeaver as the required interfaces, which are the provided interface of the components Change Request Parser, Aspect Generator and AspectWeaver respectively. In the same way, IRemoteInterface, IRemoteMethod are provided interfaces of Component Meta Model
Component and IMethodInvoker is the provided interface of Component Function Model and IRemoteMethod is the required interface of it.

Figure 4.1 Adaptable middleware model (AMM)

4.2 ARCHITECTURE IMPLEMENTATION

The component services stated in the proposed middleware were implemented through the classes specified in the class structural view of the middleware as shown in Figure 4.2. This implementation structure addresses the issues of handling multiple requirement changes, reflecting the change on multiple functions, allowing more than one component to adapt to the changes and reusing the implementation of changes. The Adapt Manager class implements the changes request received in the xml format as shown in Figure 4.3. It initiates the operations to parse the xml file and provides the change request details that includes component name, operation name and
functional change specification to generate the aspect and method wrapper for dynamic invocation of the function associated with the changes. It also triggers the dynamic weaving process to bind the aspect with the method wrapper. The method for parsing the xml file is implemented in the Change Request Parser class. Maintaining the method details and the functional change specifications for creating aspect and method wrapper is defined as the responsibilities of the Aspect Generator class. Creating the wrapper for the method associated with the change request is implemented in CFWrapper class.

The Change Request Aspect defines advice for specifying the implementation of dynamic changes. It creates the dynRequest point cut to specify the location for weaving the request and generates the unique file path to store this aspect class. The file path along with the change specification is stored in a database for reusing it to represent the similar type of changes. The dynamic structural introspection is implemented in Component Meta class through objectLookup, classLookup, methodLookup and interfaceLookup methods. Dynamic weaving using load time weaving mechanism is defined in the Aspect Weaver class. The implementation of the classes described in Figure 4.2 using Java and AspectJ platform is given in Appendix 1.
Figure 4.2 Class structural view of AMM
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://www.example.org/DynRequestSchema/">
  targetNamespace="http://www.example.org/DynRequestSchema/">
  <element name='CHANGEREQUEST'>
    <complexType>
      <sequence>
        <element name='COMPONENTS'>
          <complexType>
            <sequence>
              <element name='COMPONENTNAME' type='string' minOccurs='0'
                        maxOccurs='unbounded' />
            </sequence>
          </complexType>
        </element>
        <element name='OPERATIONS'>
          <complexType>
            <sequence>
              <element name='OPERATIONNAME' type='string' minOccurs='0'
                        maxOccurs='unbounded' />
            </sequence>
          </complexType>
        </element>
        <element name='CHANGESPECIFICATION'>
          <complexType>
            <sequence>
              <element name='CHANGEDETAILS' type='string' minOccurs='0'
                         maxOccurs='unbounded' />
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
</schema>

Figure 4.3 Change request XML schema structure
The operations involved in the functional change adaptation scenario as shown in Figure 4.4 are:

- receiving the functional change request by the Adapt Manager
- extracting the request details by sending the change request from Adapt Manager to Change Request Parser
- maintaining the change specification details in Aspect Generator for aspect implementation, which is provided by the Adapt Manager
- creating the Change Request aspect to implement the functional change using the change specification provided by the Aspect Generator
- generating the wrapper for the function associated with the change in CFWrapper on receiving the wrapper generate initiation message from Aspect Manager and getting the method details from the ComponentMeta class through performing method lookup operation
- weaving the advice of the aspect with wrapper defined in the CFWrapper using load time weaving implemented in Aspect Weaver.
Figure 4.4 Behavioral view of AMM
4.3 FORMAL ANALYSIS OF THE MIDDLEWARE

Lemma 1: An adaptable system can adapt to dynamic changes without altering and modifying it

The functionalities of the architectural components are analyzed in the view of achieving the objective stated as adapting dynamic changes without modifying and reconfiguring the system. This evaluation is performed by analyzing the behavior of the architectural components in the scenario of adapting dynamic changes as follows. Let $F$ be the function defined in a component and $\delta F$ be the change to be made dynamically on the function $F$. The change request $\delta F$ is implemented as an advice ‘AD’ in the Aspect Generator component and the change is incorporated into the corresponding function $F$ by joining the advice AD with the wrapper of the function $F$ is referred to as "WRAPPER(F)" that is generated dynamically by the Component Function Wrapper component. The location into which the advice is to be woven is specified as joint-points (JP) and the corresponding details are maintained by Aspect Generator component. Introspection is the process of providing meta-information of the service component that is defined in the Component Meta Model component. Hence dynamic adaptation of functional changes in the service component is expressed in Equation (4.1) as dynamically invoking the change $\delta F$ into the function $F$ that is described as the process of weaving the advice AD at JP of the WRAPPER(F). The operator ‘+’ is redefined to represent invocation and the operator ‘A’ is redefined to represent the binding of wrapper with join-point in Equation (4.1)

$$F + \delta F = \text{Weaving} (\text{AD}, (\text{WRAPPER}(F) \land \text{JP})),$$  \hspace{1cm} (4.1)

Weaving the aspect with the function is represented as joining the advice code with the existing functional code. Using this principle, the
expression stated in Equation (4.1) can be rewritten as follows. The operator \( \triangleright \) in Equation (4.2) represents joining of aspect with advice.

\[
F + \delta F = AD \triangleright (WRAPPER (F) \land JP),
\]  

(4.2)

The expression specified in Equation (4.2) states that the changes on a function of the component is performed by binding the advice defined in the aspect with the wrapper of the function, which clearly shows that the structure and behavior of the function associated with the change is not modified.

**Lemma 2:** Changes in a function will not affect its dependent functions if the changes are not to be reflected on those functions

A formal analysis is carried out to prove that the proposed adaptable middleware is allowing the system to reflect the changes only on the associated methods.

In this formal specification CNAME, IFNAME and MNAME are sets representing the names of the component, interface and method respectively. DYNREQ and DELTACHANGEIMPLEMENTOR are the sets representing the dynamic requirements specification and the procedures to implement the dynamic requirements respectively. JOINTPOINTSS is a set specifying the point of execution of the advice in the methods.

\[
\text{Interface} = \text{CNAME} \times \text{IFNAME}
\]

\[
\text{Method} = \text{IFNAME} \times \text{MNAME}
\]
The structure used to describe the components consists of Schema name that specifies the name of the component, the declarations including data and functions and invariants includes property rules associated with the data and functional specifications, as shown as in Figure 4.5. The property rules associated with the method specification describes the pre and post states of the method using the domain and range specification.

```
-----Schema Name-------
declarations
invariants
```

**Figure 4.5 Schema structure**

The components of the proposed model are described using the schema structure as stated in Figure 4.5. The Component Meta Model schema specified in Figure 4.6 describes the functions interfaceLookup and methodLookup. The interfaceLookup takes the component name as input from the domain set CNAME and returns interface name from the range set IFNAME. The methodLookup is described along with the domain set from which it has to take input and range set to specify the output range.

```
-----Component Meta Model-------
interfaceLookup : cname → ifname
methodLookup : ifname → mname

| dom interfaceLookup = | ran interfaceLookup = |
| cname, cname ∈ CNAME | interface, ifname ∈ Interface |
| dom methodLookup = | ran methodLookup = |
| ifname, ifname ∈ IFNAME | mname, mname ∈ Method |
```

**Figure 4.6 Component meta model schema**
A standard function for executing the method is named as Execute:
Inparameter → Outparameter. The Component Function Wrapper provides
wrapper service, which is described in Figure 4.7. The Wrapper method takes
the method name as an input from the domain MNAME and executes that
method to generate the output as specified in the Outparameter of the method.

```
<table>
<thead>
<tr>
<th>Component Function Wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrapper : MNAME → ExecutionResult</td>
</tr>
</tbody>
</table>
```

*Figure 4.7 Function wrapper schema*

The Aspect Generator Schema describes the way of defining the
aspect with the point-cut named as dynRequest as shown in Figure 4.8. The
dynRequest is the combination of the advice and it represents the code
associated with the changes in the dynamic requirements as the
DELTACHANGETEMPLATEMENTOR and the location to which the advice is to
be weaved as the JOINPOINT.

```
<table>
<thead>
<tr>
<th>Aspect Generator</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynRequest : DELTACHANGETEMPLATEMENTOR x JOINPOINT</td>
</tr>
</tbody>
</table>
```

*Figure 4.8 Aspect generator schema*

The Aspect Weaver Schema depicted in Figure 4.9 describes the
association between the Wrapper and dynRequestMethod, which is named as
dynamicWeaver. The Wrapper method is the member of the Remote Method
Invoker class and dynRequestMethod is the member of the Aspect Generator
class.
### Aspect Weaver

<table>
<thead>
<tr>
<th>dynamicWeaver : Wrapper ↔ dynRequestMethod</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom dynamicWeaver = Wrapper ε Remote Method Invoker</td>
</tr>
<tr>
<td>ran dynamicWeaver = dynRequestMethod ε Aspect Generator</td>
</tr>
</tbody>
</table>

**Figure 4.9 Aspect weaver schema**

The property of the adaptable middleware stated in the lemma is proved using the schema descriptions provided for the middleware components. Let \( m \) represent the method to be modified, \( \delta m \) represent the changes in the requirements, \( \text{wm} \) represent the Wrapper method of the Component Function Wrapper component as shown in Figure 4.7 and \( \text{advice} \) represent the dynRequest of the Aspect Generator component as shown in Figure 4.8. The operator \( + \) represents invoke operation and \( \bowtie \) represents bind operation.

In this work, adaptation function is defined as in Equation (4.3).

\[
f : (m + \delta m) \rightarrow (\text{wm} \bowtie \text{advice})_x \tag{4.3}
\]

The function \( f \) shows that the inclusion of the dynamic requirement \( (\delta m) \) into the method \( (m) \) is achieved through integrating the advice with the wrapper method. Let \( M_x \) be the method to be modified. The relation \( R \) is defined to show the association between the method \( M_x \) and its dependent methods \( M_y \), which is shown in Equation (4.4).

\[
R : M_x \leftrightarrow M_y, \text{ where } M_x \text{ is defined as dependents of } M_x \tag{4.4}
\]
According to the adaptation function, changes to the method $M_\kappa$ is performed through Wrapper of the $M_\kappa$ ($wm_\kappa$). Hence, the changes in the method $M_\kappa$ will not affect the dependent methods $M_\lambda$. This discussion is expressed in Equation (4.5).

$$(M_\kappa + \delta m_\kappa) = (wm_\kappa \oslash \text{ advice}) \implies (M_\kappa + \delta m_\kappa) \not<=> M_\lambda \quad (4.5)$$

4.4 DEMONSTRATION OF THE MIDDLEWARE

The banking transaction system described in Section 3.5 is constructed with the user interface management component named as UI Management, Authentication component and Banking Transaction component. UI Management provides services for formatting the content information and for defining and organizing the display controls. Authentication component is responsible for validating the users and their account details while performing the transactions. Banking Transaction component provides support for performing various operations including opening and closing the account in a bank, deposit, withdrawal and transfer of money and paying bills through the bank. This system is enriched with adaptability feature by integrating it with the proposed adaptable middleware as shown in Figure 4.10.
Figure 4.10 Adaptable banking transaction system structure
The changes occurred in the authentication, user interface management and transaction service scenarios as stated in Section 3.5 are captured as dynamic aspect through the following steps.

1. Submitting the information associated with the changes in the XML file and its structure is shown in Figure 4.3.

2. XML file is received by the Adapt Manager and performs the following operations. For instance, a request for change in the authentication scenario is shown in Figure 4.12. The behavior of authentication service before incorporating the change is shown in Figure 4.11.

3. Adapt Manager redirects the XML file to Change Request Parser by calling xmlParser function defined in Change Request parser for parsing it.

3.1 It parses XML file and returns the component name, operation/function name and change details to Adapt Manager. The parsing result of the change in the authentication strategy is shown in Figure 4.13.

4. Adapt Manager invokes the Aspect Generator object for creating aspect class to define the change.

4.1 Aspect Generator gets the change-related information from Aspect Adapter for implementing the changes in the Change Request Aspect class.

4.1.1 Change Request Aspect class is defined as a template with the point cut named as „dynRequest”, binding method as methodWrapper
and advice for specifying the implementation information of the dynamic changes. The advice will take a different implementation code for each change, whereas other parts of the class will remain same. The Change Request Aspect class to represent the authentication strategy change is shown in the Figure 4.14.

5. Finally Adapt Manager invokes the call to initiateWrap method defined in CFWrapper for generating wrapper to invoke the function defined in the application component associated with the dynamically proposed change.

5.1 For instance, to adapt the changes AC1, AC2 and AC3 specified in the Authentication scenario as stated in Section 3.5, the userAuthentication method defined in the Authentication component is invoked by extracting the method details using Component Meta services.

Similarly for change MTC1 in Money Transaction Scenario, the deposit, withdrawal and fundTransfer functions in Banking Transaction Component are invoked in the order of operation in execution through the wrapper method.

5.1.1 CFWrapper class sends methodLookup message to Component Meta class for getting the method details by passing component and method name. Component Meta extracts the details about the interfaces that are implemented in the component.
5.1.2 Component Meta class extracts the method name, input parameters and returns parameter information for the methods that are implemented in the interface, and returns them as Method objects to CFWrapper. For instance, while adapting AC1 change, accountAuthentication and userAuthentication methods are returned. Also, for adapting MTC1, open/close account, deposit, withdrawal, fundTransfer, billPayment, EMI.Payment and principalPayment methods are returned to CFWrapper.

5.2 CFWrapper selects the function associated with the change by matching operation name specified in the XML file with the method name specified in Method object, and applies invoke call to perform method invocation. For instance, userAuthentication is selected and invoked for AC1 change and deposit, withdrawal and fundTransfer functions are selected and invoked for MTC1 change in a sequential way.

6. Finally, Adapt Manager sends aspectWeave message to Aspect Weaver to dynamically bind the advice defined in the Change Request Aspect class with the methodWrapper of the CFWrapper class. The result of strategy change invocation in the authentication service is shown in Figure 4.15.
The user id is suganthi.
The system generated password is suganthi113752.
The user validity is confirmed with userid and password.

Figure 4.11 Authentication result before change

```xml
<?xml version="1.0" encoding="utf-8"?>

<CHANGEREGUIEST>

<COMPONENTS>

  <COMPONENTNAME>"Authentication"</COMPONENTNAME>

  <OPERATIONNAME> "User Validation"</OPERATIONNAME>

  <CHANGESPECIFICATION>"include bio-metric verification"</CHANGESPECIFICATION>

</COMPONENTS>

</CHANGEREGUIEST>

Figure 4.12 Authentication strategy change request
Figure 4.13 XML parser result
4.5 SUMMARY

The proposed adaptive middleware architecture abstracts the adaptability solution through components, which makes it as a general solution. This generic feature allowed the functions of the adaptive middleware components to have different platforms for implementation. Most of the existing adaptability solutions introduced adaptability features on the middleware models such as COM, CORBA and EJB. Hence the applications developed using these models can only avail the adaptability facility defined with them. This limitation has been addressed in the proposed adaptive middleware model. The techniques used in the proposed model are defined using aspect-oriented approach, introspection and adapter, which is easy to understand and implement. Representing details about the changes in the requirements using user-friendly format rather than using xml format and implementing the changes in aspects using meta-application knowledge are to be considered in the future work. This proposed model mainly focuses on accommodating changes in the requirements. Hence, it can be extended to adapt to variations in the environment in the future.