Abstract

As a solution to eliminating component mismatches, this paper presents a generative aspect oriented approach to component adaptation. The approach enjoys high level of automation and capability of deep level adaptation, which is achieved in an aspect-oriented component adaptation framework by generating and then applying the adaptation aspects under designed weaving process. The aspect generation mechanism facilitates the creation of adaptation aspects that support specific adaptation requirements. An expandable repository of reusable adaptation aspects has been developed based on the proposed two-dimensional aspect model. A prototype tool is built to as a leverage of the approach.

Keywords—Aspect-Oriented Programming, Generative Component Adaptation, Aspect Generation, Software Reuse.

1. Introduction

Component-Based Development (CBD) has been proved as an effective technology in supporting community-wide reuse of software assets [13]. Recently, the methodology of CBD has been expanded to build more challenging and modern systems, such as dependable embedded software and web services, which consequently imposed more rigid
requirements on the methodology itself [6][7].

Due to the availability of components and the diversity of target applications, in many cases mismatches between pre-qualified available components and the specific reuse requirements in particular applications are inevitable and have been a major hurdle of wider component reusability and smooth component composition. These mismatches may occur in aspect of functionality or quality such as performance, security and safety, and degrade the target component-based system severely. Component adaptation has been researched over the years as a key solution to the above problem [3][9][11]. Due to the complex nature of the mismatch problem, available approaches are either only capable for adaptation at simple levels such as wrappers [3], or inefficient to use due to the lack of automation in their adaptation process. To assure the quality of the target component-based software, more efficient automated adaptation mechanisms are in need to eliminate component mismatches.

In this paper, a generative aspect-oriented component adaptation approach is proposed to achieve automated component adaptation at a rather deep level, particularly aiming at eliminating mismatches in non-functional aspects such as system dependability. The approach is based on the successful points in a few technologies, i.e., Aspect Oriented Programming [10][12][14][18], Software Product Line [2][5] and Generative Component Adaptation [1][4]. In the approach, component adaptation is carried out within an aspect-oriented component adaptation framework by generating and then applying the adaptation aspects under designed weaving process. The generation absorbs the variation concept of software product line and ensures the suitability of adaptation aspects for the specific adaptation requirements of the aimed reuse context. Compared with traditional AOP, the
weaving process of aspects in our approach supports more complex control flow, i.e., not only sequence, but also switches, synchronization and multiple threads, in order to deal with adaptation in more complicated environments such as concurrent and large scaled applications. To facilitate the reusability of adaptation knowledge, based on generative programming techniques, an expandable repository of reusable adaptation aspects is developed based on the proposed two-dimensional aspect model. A prototype tool is built as the leverage of the approach.

The reminder of the paper is organized as follows: Section 2 introduces the generative aspect model (repository). Section 3 describes the aspect-oriented generative component adaptation process. Section 4 presents the process based component adaptation specification. Section 5 introduces the prototype tool, and section 6 presents an example to demonstrate the approach. Section 7 discusses the related work. Finally, section 7 presents the conclusion.

2. Generative aspect model

In our approach, the adaptation knowledge is captured in aspects and aims to be reusable in various adaptation circumstances. To achieve automatic generation of the adaptation aspects, a two dimensional aspect model is developed and in practice an aspect repository is built as an embodiment of the above model.

Figure 1 presents the views of an aspect from two different dimensions: abstraction view and component view. Figure 2 presents the aspect generation process.
2.1 Abstract view

In the proposed aspect model, an aspect is defined at three abstraction levels, which constitutes the abstraction view.

\[
\text{ASPECT ::= \{AAF | AF | AInst \}, where}
\]

AAF stands for Abstract Aspect Frame, which defines the structure of the aspect; AF stands for Aspect Frame, which is an instantiation of AAF in a specific adaptation circumstance; AInst stands for Aspect Instance, which is an instantiation of AF on a specific AOP platform.

AAF is the fundamental and the most abstract level of aspect definition in the model. As an XML schema file, AAF is used to define the structure of different types of aspects. According to the functionality, AAF forms a hierarchical structure that reflects functional variations of different adaptations. Adaptation aspects are modelled into different types, for example, logging, caching, authentications, etc. Each aspect type is then refined into a group of sub-types. For example, aspects about authentication may consist of operating-system-based authentication and database based authentication.

An example of the AAF for a \textit{DB connection pool} aspect is given in section 6.

Each AAF may have many Aspect Frames (AFs). AF is the second abstraction layer in aspect definition. An AF is an instance of the related AAF in a specific adaptation circumstance. Compared with its AAF, an AF has the details of a concrete aspect populated into it by assigning values to the parameters. User interaction is required to create an AF from an AAF. Defined in XML format, AF is independent from concrete AOP platforms such as AspectJ.
An example of the AF for a logging aspect is given in section 6.

An AF is not executable until it is mapped onto a concrete AOP platform. The result of this mapping is a family of Aspect Instances (AInst) based on various AOP platforms. An Aspect Instance is executable and specific to a concrete AOP platform, and it reflects platform variations of an aspect on different AOP platforms. The agent to generate AInst from their AF is called Semantic Interpreter. The generation process is fully automatic. An example of AInst is shown in section 6.

2.2 Component view

The component view presents the structural elements inside an aspect.

\[ \text{ASPECT} = (\text{CS}, \text{V}) \]

\[ \text{CS} = (\text{P}, \text{A}) \]

\[ \text{P} = (\text{N}, \text{T}, \text{O}) \]

\[ \text{N} = \text{Name of the pointcut} \]
\[ \text{T} = \text{Applying time of the pointcut} \]
\[ \text{O} = \text{Object that the pointcut applies to} \]

\[ \text{A} = \text{Declaration of Advice} \]

\[ \text{V} = (\text{DATA}, \text{SI}) \]

\[ \text{DATA} = \text{Data of variations} \]
\[ \text{SI} = \text{Semantic Interpreters} \]

- **CS** is the Common Structure of aspects. CS defines the common structural elements that any aspect will have despite its functionality or implementation platform. CS
consists of two sub-elements: Pointcut (P) and Advice (A). All aspects have the same CS at AAF level no matter how different these aspects are in functionality and implementation platform.

- **P** is the specification of Pointcut, which includes three parts: N, T and O. N defines the name of the pointcut, T defines the applying time of the pointcut, and O is the object that the pointcut applies to, includes the component and method signatures.

- **A** is the declaration of Advice, which contains the basic information of the advice, e.g. when the joint point will be inserted into the target component.

- **V** is Variations, which defines the variations of different aspects. V consists of two parts: the data of the variations and the processing logic the actions of the aspect. V is aspect specific information. V consists of two sub-elements: Data and SI.

  - **Data** refers to the parameter values of the variations. For example, in a logging aspect Data includes the logging message to be saved, and the file name used to save the message, whilst in a database connection pool aspect the capacity of the pool is part of its variation parameter.

  - **SI** is the Semantic Interpreters of the aspect, which specifies the actions to be carried out when the aspect is applied.

2.3 Aspect generation process

In the proposed aspect model, the generation process of an aspect can be viewed as the refinement of an AAF into an AF and finally into an AInst. If we define \( \vDash \) as refinement, then

\[
\text{AAF} \vDash \text{AF} \vDash \text{AInst}
\]

Syntactically, AAF is the definition of a set of XML Schemas, which defines the
structure of an aspect. AF is an XML file with the above schemas populated with data. User interaction may be needed when refining an AAF to an AF. AInst is the final binary or source code of the aspect. The transformation from an AF to an AInst is done by the code generator which takes the Data and SI in Variations (V) as input. The process is automatic.

As shown in figure 2, a product family of the adaptation aspects is designed to achieve high reusability. All aspects are defined at three abstraction levels: AAF, AF, and AInst. These three abstraction levels of aspects facilitate the reusability of adaptation aspects as they realize different variations of these aspects. Functional variations are achieved by a hierarchical classification of aspects at AAF level. Parameter variations reflect the refinement of aspects in specific adaptation circumstances, which is achieved by the determination of the value of CS and V in AF. Platform variations reflect the implementation of aspects on particular software platforms, which is done by mapping the aspect to particular software platforms.

With the basic definition in figure 1, the tuple (CS, V) is refined across the three abstraction layers in both format and contents.

At AAF level, the tuple is XML schema definition. CS consists of the elements to describe the common structure of an aspect. V consists of the elements to declare the parameters of variations of the aspect (Data), and the processing logic of the aspect’s actions (SI), which is defined in XSL.

At AF level, conforming to the structure definition in AAF, the tuple is filled with the
XML format data specific to an adaptation circumstance. CS will have the data such as the aspect name, the signature of the particular component and method(s) on which the aspect to be applied. V will be filled with parameter values which determine the specific adaptation circumstances in which the aspect is applied.

At AInst level, a concrete executable aspect is generated automatically with the code generator, that is the common structure (CS), variation data (Data) and aspect semantics (SI) are finally mapped into executable programme code. The first step is to map the processing logic in SI to the platform-independent Intermediate Language (IL), and the second step if to translate the IL code to a platform specific language, such as AspectJ.

3. Aspect-oriented generative adaptation process

Figure 3 describes the process of the proposed aspect-oriented generative component adaptation, which involves the creation and application of suitable aspects with the aspect model defined in section 2. We presume that in a component based system, a component has been found potentially suitable for an application, but some mismatches still exist, therefore, the application developer wishes to have the component adapted.

The mismatch will be eliminated by applying aspect-oriented adaptation to the original component. At start, the component is analyzed with the component analyzer, which analyzes the source or binary code of the component and extracts component specification information, e.g. class names and method signatures. The component specification will be used to guide component adaptation. If the component already has well defined specification, this step can be skipped.
Then based on the adaptation requirements, a Process-based Component Adaptation Specification (PCAS) will be created by selecting aspects defined at the abstraction level of Abstract Aspect Frames (AAF) and define the weaving process of these aspects. The selection of aspects is actually the process to determine functional variation of a specific adaptation. The composition of PCAS is supported by an interactive IDE called PCAS Editor, which supports both graphical and XML source view of the PCAS.

A PCAS is an XML formatted document, which includes the details of component adaptation, such as the target component, the weaving process, and the abstract aspects to be applied. In a PCAS, sequence and switch structure are supported to achieve flexible adaptation on components. In PCAS, the adaptation process is depicted with only the ID of the selected aspects. Full details of the aspects are still kept in Aspect Repository.

Based on PCAS and the lower level aspect definition, namely Aspect Frame (AF) in the aspect repository, executable aspects instances (AInsts) are generated by the aspect generator according to different AOP implementation specifications. As a result, platform variation is achieved during aspect generation. The input for the aspect generator is AF and the output is AInsts.

The aspect repository is an embodiment of the proposed generative aspect model. Reusable aspects are defined at three abstraction levels and kept in the repository as AAF, AF, and AInst. The reusable assets in the repository include both primitive and composite aspect types, which comes from the adaptation process in PCAS.
The aspect manager is a tool to manage reusable aspects in the aspect repository, and to present graphical views of aspects at various abstraction levels.

The generated executable aspects are finally applied to the component by the aspect weaver. A new adapted version of the component is then created through aspect weaving. Since current AOP platform like AspectJ does not support complicated flow control such as switch in weaving process, post-processing is applied to enable process-based weaving in our framework.

4. Process based component adaptation specification (PCAS)

To satisfy the adaptation requirements for a particular reuse context, it often requires performing complex adaptation to multiple components with a set of generated aspects applied to these components under a specially designed process containing conditions, synchronization and other flow controls. Process-based Component Adaptation Specification is developed to describe the above complicated adaptation details. Finally implemented in XML schema, PCAS is defined with the following tuple:

\[
\text{PCAS} = (C, A, P), \quad \text{where}
\]
\[
\begin{align*}
C &= \text{Component} \\
A &= \text{Aspect} \\
P &= \text{Process}
\end{align*}
\]

- \(C\) defines the components on which aspects apply.
- \(A\) defines the aspects to be applied in the process, including the id, type and level of the aspects.
**P** defines process control, including execution mode ("Sequence", "Switch", and "Case") and the guard condition. Because multiple aspects may access the same resource, e.g., a file, PCAS supports synchronization.

The PCAS definition is implemented in XML schemas. The default structure is given in table 1.

```
<AOP-Process name="xxxx" xmlns="http://www.dcs.napier.ac.uk/2005/PCAS">
  <Container name="xxxx">
    <Sequence>
      <Apply-aspect class="xxxx" method="xxxx"
        aspect_id="xxxx" aspect_level="xxxx"
        aspect_type="xxxx" af_id="xxxx"
        af_name="xxxx" synchronized="xxxx"
        comment="xxxx"/>
      <Switch expr="xxxx">
        <case value="xxxx">
          <Apply-aspect class="xxxx" method="xxxx"
            aspect_id="xxxx" aspect_level="xxxx"
            aspect_type="xxxx" af_id="xxxx"
            af_name="xxxx" synchronized="xxxx"
            comment="xxxx"/>
        </case>
        <case value="xxxx">
          <Apply-aspect class="xxxx" method="xxxx"
            aspect_id="xxxx" aspect_level="xxxx"
            aspect_type="xxxx" af_id="xxxx"
            af_name="xxxx" synchronized="xxxx"
            comment="xxxx"/>
        </case>
      </Switch>
    </Sequence>
  </Container>
</AOP-Process>
```

Table 1. Process based component adaptation specification

If a PCAS is found common and reusable in the future, its process control part can be regarded as a composite aspect type. Composite aspects are supported in AAF level to achieve advanced reuse in typical aspect using cases.

To implement PCAS in weaving process, a post-weaving technique is developed. The post-weaving tool gets class files for aspects generated by AOP platform such as AspectJ as input, and then modifies those class files to generate new class files that support
complicated flow control and synchronization according to PCAS.

5. The prototype tool

A CASE tool has been developed to scale up the proposed approach. With this tool, component developers define aspect weaving process by drag-and-drop in a graphical interface, they select candidate aspects and fill in necessary details of CS and V. The semantic interpreter will generate AInsts automatically. According to the defined PCAS, Aspect Weaver will complete the aspect weaving and generate adapted components.

The tool includes the following parts: 1) PCAS Editor, which provides an edit environment for PCAS both in graphical interface and at XML level. A screen dump is shown in figure 4. 2) Aspect Manager, which supports the management of reusable aspects in Aspect Repository and the graphical view of different levels of aspects. Aspects at different levels can be created, removed, and edited in Aspect Manager, either in the graphical user interface, or at XML level. A screen dump of Aspect Manager is shown in figure 5. 3) Component Analyzer, which analyzes component and gets necessary information such as the class names and method names, for component adaptation. 4) Aspect Generator: based on AFs and corresponding Semantic Interpreters, executable aspect instances will be generated by Aspect Generator. The generated executable aspects will be saved into aspect repository as AInsts. 5) Aspect Weaver, which is used to generate new components by weaving generated AInsts into original components.

6. Case studies

Case studies have been done to verify the approach, in terms of its capability of building
highly reusable aspects across various AOP platforms and providing a complex flow control of weaving process. In this section, we use the following case study to demonstrate how PCAS works and how to generate an executable aspect by mapping through the different abstraction views of the aspect in our framework.

As shown in figure 6, the case study regards with an on-line testing component, which was developed by Component Source, a software company selling COTS components. The IT department of a university planned to build their online assessment system and bought the component for integration as part of the system. However, they identified that the large student number would impose heavy access load and make the system performance poor.

The development team decided that prior to integration of the online testing component three actions should be done to adapt the component. First, a database connection pool is to be introduced to the online testing system to improve system performance. Then, logging is used to monitor the usage of the connection pool. Finally, based on the logging information, the connection pool is tuned to achieve the best performance with reasonable resource cost such as memory consumption, by adjusting the parameters, including the capacity of connection pool and the time of expire of a connection instance.

To meet the reuse requirements, the following three aspects are applied to the component to implement the above adaptation actions, namely database connection pool, logging if connection pool reaches its maximum capacity, and logging if connection pool does not reach its maximum capacity. These adaptation actions are then described in a Process-based Component Adaptation Specification (PCAS) shown in table 2. The specification is created
with the PCAS Editor by finding appropriate AAFs, i.e., either primitive types or composite types of aspects, and putting these AAFs into an adaptation process. Functional variation of adaptation is implemented through the composition of PCAS. In table 2, the PCAS specifies to weave three aspects into the on-line testing component, namely DBConnectionPool (dbp-1), logging under full pool capacity (dbp-logging-1), and logging over full pool capacity (dbp-logging-2). DBConnectionPool (dbp-1) will be applied first, sequentially followed by a switch condition: applying dbp-logging-1 if the pool has not reached its full capacity, or applying dbp-logging-2 if the pool has reached its full capacity. The id, type, AF name, synchronization mode of the aspects are defined. The joining point is defined as “getConnection” method in the component.

```xml
<?xml version="1.0"?>
<AOP-Process name="Aspects on On-line testing"
xmlns="http://www.dcs.napier.ac.uk/2005/PCAS">
<Container name="Connection pooling and logging on database connections">
<Sequence>
<Apply-aspect
class="java.sql.DriverManager,java.sql.Connection"
method="getConnection"
aspect_id="030001"
aspect_level="primitive"
aspect_type="DBConnectionPool"
af_id="03000101"
af_name="dbp-1"
synchronized="false"
comment="Add all DB connections into the pool"/>
<Switch expr="dbp-1.getDBPStatus()">
<case value="true">
<Apply-aspect
class="java.sql.DriverManager,java.sql.Connection"
method="getConnection"
aspect_id="010001"
aspect_level="primitive"
aspect_type="logging"
af_id="01000101"
af_name="dbp-logging-1"
synchronized="true"
comment="Tracing while DB connection pool does not reach its capacity"/>
</case>
<case value="false">
<Apply-aspect
class="java.sql.DriverManager,java.sql.Connection"
method="getConnection"
aspect_id="010001"
aspect_level="primitive"
aspect_type="logging"
af_id="01000102"
synchronized="true"
comment="Tracing while DB connection pool does not reach its capacity"/>
</case>
</Switch>
</Sequence>
</Container>
</AOP-Process>
```
The specification in PCAS is at a rather overview level and does not contain the details of individual aspects. Developers need to provide parameter values for each aspect. Common AFs can be saved into Aspect Repository for further reuse. In this example, three AFs will be generated for each of the above aspects accordingly. Due to the structural similarity of AFs of different aspects, we only give the AAF for DB connection pool in table 3, and the AF for logging while DB connection pool does not reach its capacity in table 4 as an example.

```
<xs:element name="Aspect">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="CommonStructure" />
      <xs:element ref="Variation" />  
      <xs:attribute name="name" type="xs:NMTOKEN" use="required" />
    </xs:complexType>
  </xs:element>
  ......  
</xs:element>
```

```
<xs:element name="CommonStructure">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="PointCuts" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<xs:element name="Variation">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Capacity" />
      <xs:element ref="IdleTime" />
      <xs:attribute name="type" type="xs:NMTOKEN" use="required" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<xs:element name="When">
  <xs:complexType mixed="true" />
```

Table 2. A Process-based Component Adaptation Specification
In table 3, the AAF consists of the frame definition of CS (Common Structure) and V (Variation) of the DBConnectionPool aspect. CS defines the structure of a basic aspect, e.g. name of an aspect and the details of an advice. V defines the distinct characters of different aspects, e.g. the capacity of db connection pool and the idle time of each connection in db connection pool aspect.

Table 3. The AAF of the DB connection pool aspect
Table 4. The AF of the logging aspect

In table 4, the AF defines the detailed parameter values of the logging aspect (*dbp-logging-1*). The basic information of aspect e.g. pointcut and advice is defined in CS and the distinct characters (variations) for logging aspect e.g. logging message, date, and time are defined in V.

```java
import java.io.*;
import java.util.*;
import org.aspectj.lang.*;

public aspect dbp-logging-1 {
  pointcut dbp-pointcut-1():execution(* java.sql.DriverManager.java.sql.Connection .getConnection(..));
  after():dbp-pointcut-1() {
    Calendar cal = Calendar.getInstance();
    try {
      FileWriter fw = new FileWriter("D:\ On-lineTesting\logs\dbp.log ", true);
      PrintWriter pw = new PrintWriter(fw);
      pw.print("Access to DB connection pool without reaching its capacity on ");
      pw.print(cal.get(Calendar.YEAR) + ".");
      pw.print(cal.get(Calendar.MONTH) + ".");
      pw.print(cal.get(Calendar.DAY_OF_MONTH) + ", ");
      pw.print("at ");
      pw.print(cal.get(Calendar.HOUR) + ":");
      pw.print(cal.get(Calendar.MINUTE) + ":");
      pw.print(cal.get(Calendar.SECOND));
      pw.println();
      pw.close();
    } catch(Exception e) {
      System.out.println("Error occurred: "+ e);
    }
  }
}
```
Table 5. A simple Aspect Instance of the logging aspect

From the AF of logging aspect in table 4, Aspect Generator generates an aspect instance (AInst) that is specific to a selected AOP platform. As shown in table 5, the generated AInst is based on AspectJ as the target platform. It is made of source code statements, which includes two parts: CS and V.

The Aspect Weaver weaves the generated aspect instances into the original component according to the PCAS. The final adapted component source code is invisible to the developer. By deploying the adapted component, the targeted users’ requirements regarding to system performance is fulfilled.

In conclusion, compared with conventional AOP platform such as AspectJ, this case study has shown the distinct features of the proposed approach and tool. These features are not supported in conventional AOP platforms including AspectJ. For example, complex flow control of weaving process has been achieved; a multiple abstraction leveled two dimensional model of aspects has been developed for automatic aspect generation and reuse. Moreover, the aspects (AAFs and AFS) are universally applicable on any AOP platform, e.g., the tool generated and weaved the AInst of the logging AF in table 4 on AOPHP by employing appropriate semantic interpreters.

7. Related work

7.1 Binary Component Adaptation
Binary Component Adaptation (BCA) [11] has been proposed by R. Keller and U. Hölzle to support component adaptation in binary form and on-the-fly (during program loading). BCA rewrites component binaries before (or while) they are loaded, requires no source code access and guarantees release-to-release compatibility. That is, an adaptation is guaranteed to be compatible with a new binary release of the component as long as the new release itself is compatible with clients compiled using the earlier release.

However, together with the binary code adaptation, especially with “online” (on-the-fly) adaptations, extra processing time is required. As a result, the load-time overhead is a major problem. Consequently, when more adaptation processes are required, the load-time will be the bottleneck of the system performance.

7.2 Superimposition

Superimposition [3] is a novel black-box adaptation technique proposed by J. Bosch at University of Karlskrona/Ronneby. Software developers are able to impose a number of predefined, but configurable types of functionality on reusable components. The notion of superimposition has been implemented in the Layered Object Model (LayOM), an extensible component object language model. The advantage of layers over traditional wrappers is that layers are transparent and provide reuse and customizability of adaptation behaviour.

Superimposition uses nested component adaptation types to compose multiple adaptation behaviours for a single component. However, due to lack of component information, modification is limited at simple level, such as conversion of parameters, and refinement of
operations. Moreover, with more layers of code imposed on original code, the overhead of the adapted component increases heavily, this degrades system efficiency.

7.3 SAGA project

Scenario-based dynamic component Adaptation and GenerAtion (SAGA) [9] developed a deep level component adaptation approach with little code overhead through XML-based component specification, interrelated adaptation scenarios and corresponding component adaptation and generation.

SAGA project focused mainly on generative component adaptation at binary code level, i.e., the adapted part of the component will be generated as new blocks of binary code and these blocks will then be composed with other unchanged blocks of code to form a new adapted component.

SAGA project achieved deep adaptation with little code overhead in the adapted component; however, automation is a challenge in the SAGA approach because it is always complex to generate blocks of code according to scenarios and original component code. To reach high automation, a large set of adaptation rules and domain knowledge has to be developed to support the process, and probably the application domains have to be restricted as well.

7.4 Aspectual Component

To achieve reusable aspects, Karl Lieberherr et al. introduced the concept of Aspectual Components [8]. In Aspectual Components, aspects are specified independently as a set of
abstract join points. They believe that aspect-oriented programming means expressing each aspect separately, in terms of its own modular structure. Using this model, an aspect is described as a set of abstract join points which are used when an aspect is combined with the base-modules of a software system. In this way, the aspect-behaviour is kept separate from the core components, even at runtime.

It is distinguished between components that enhance and crosscut other components and components that only provide new behaviour. An aspectual component has a provided and a required interface. Connectors connect the provided and required interfaces of other components. The connection process starts with a level-zero components consisting of very simple class definition.

7.5 JAsCo

JAsCo [15][17] is an aspect based research project for component based development, in particular, the Java Beans component model. JAsCo combines the expressive power of AspectJ [12] with the aspect independency idea of Aspectual Component. The JAsCo language introduces two concepts: aspect beans and connectors. An aspect bean is used to define aspects independently from a specific context, which interferes with the execution of a component by using a special kind of inner class, called a hook [17]. Aspect beans can be reused and applied upon a variety of components. A connector allows specifying precedence and combination strategies between the aspects and components.

However, JAsCo is not suitable for specific modification requirements since it does not provide a mechanism for conducting users’ requirements. In addition, the way to apply
aspects on target components / systems is based on traditional AOP process, and therefore, may result in lower readability, maintainability and performance. Moreover, the current implementation of JAsCo has been bounded to Java, which limits its usability.

7.6 XVCL

XVCL (XML-based Variant Configuration Language) [16] is a general purpose mark up language for configuring variants in programs and other types of documents. XVCL is capable of injecting changes, according to pre-defined plans, into programs represented as a hierarchy of highly parameterized meta-components or templates. Meta-component parameters may be as integer or string values, or complex as a hierarchy of other meta-components. XVCL uses a “composition with adaptation” mechanism to instantiate parameters and to generate concrete programs from generic meta-component architectures. Many template engines have been proposed to tackle specific problems, in specific domains. XVCL is a language, problem and domain independent template based generative engine.

However, in each x-frame in XVCL, the code templates and variables to support software product lines are mixed together. This has limited the reusability of the composition knowledge recorded in x-frames. Initially designed for general purpose application, XVCL does not support the proposed three-leveled aspect model and misses specific features for aspect oriented software engineering, hence cannot be used to achieve a high level of automation and good appropriateness in the aspect generation process.
7.7 Summary

Due to the diversity and level of component mismatch, available component adaptation approaches are either only capable for adaptation at simple levels, such as Superimposition, or inefficient to use due the lack of automation in their adaptation process, such as Binary Component Adaptation, SAGA and XVCL. In contrast, the proposed approach achieves deep-level highly automated component adaptation via automatic aspect generation and complex process based weaving.

Some AOP based frameworks have been developed to achieve reusable and dependable components, such as Aspectual Components and JAsCo. However, an AOP platform independent framework is still desired in a heterogeneous distributed environment to solve crosscutting problem since a common model for AOP is still missing. Furthermore, current AOP techniques only support weaving aspects sequentially, such as AspectJ. To cope with complex adaptation, it often requires weaving aspects in more sophisticated control flow, e.g. dynamically deciding whether to invoke a particular aspect, and synchronizing in multi-thread applications. The proposed approach and tool provide a solution for platform independent aspects and their weaving under sophisticated control flow.

8. Conclusions

Despite the success of component-based reuse, the mismatches between available pre-qualified components and the specific reuse context in individual applications continue to be a major factor hindering component reusability and smooth composition. The work presented in this paper is based on the observation that existing reuse approaches and tools
are weak in providing a mechanism to adapt components at adequately deep level and meanwhile with sufficient automation. The aspect-oriented nature of our approach makes it particularly suitable for the improvement of non-functional features of the target component-based software, such as dependability and performance.

The proposed approach applies aspect-oriented generative adaptation to targeted components to correct the mismatch problem so that the components can be integrated into the target application smoothly. Automation and deep level adaptation are the benefits of the approach. This is achieved with the following key techniques in an aspect-oriented component adaptation framework: 1) the generation of adaptation aspects based on specific adaptation requirements; 2) the advanced aspect weaving process definition mechanism that supports switch and synchronization; 3) an expandable repository of reusable adaptation aspects based on a two-dimensional aspect model.

The benefits of the approach include deeper adaptability, especially in non-functional aspects, higher automation and therefore smooth component integration and wider reusability. As consequence, the target component-based software will have better performance. Our case studies have shown that the approach and tool are promising in solving the mismatch problem.

9. References

Fig. 1. A two-dimensional view of ASPECT

Fig. 2. The three leveled aspect generation process

AAP: Abstract Aspect Frame
AF: Aspect Frame
AInst: Aspect Instance
CS: Common Structure
\( \forall \): Variations
Fig. 3. The generative aspect-oriented component adaptation framework

Fig 4. A screen dump of PCAS Editor
Fig. 5. A screen dump of Aspect Manager

Fig. 6. On-line testing component