Achieving Smooth Component Integration with Generative Aspects and Component Adaptation

Yankui Feng, Xiaodong Liu and Jon Kerridge

School of Computing, Napier University, Edinburgh, UK
{y.feng, x.liu, j.kerridge}@napier.ac.uk

Abstract. Due to the availability of components and the diversity of target applications, mismatches between pre-qualified existing components and the particular reuse context in applications are often inevitable and have been a major hurdle of component reusability and successful composition. Although component adaptation has acted as a key solution of eliminating these mismatches, existing practices are either only capable for adaptation at a rather simple level, or requires too much intervention from software engineers. This paper presents a highly automatic approach to component adaptation at adequately deep level. The adaptability and automation is achieved in an aspect-oriented component reuse framework by generating and then applying the adaptation aspects under designed weaving process according to specific adaptation requirements. An expandable library of reusable adaptation aspects at multiple abstraction levels has been developed. A prototype tool is developed to scale up the approach.

1. Introduction

Component-Based Development (CBD) has been proved as an effective technology in supporting community-wide reuse of software assets [14]. Under the methodology of CBD, both Commercial-Off The Shelf (COTS) [7] components and in-house components can be integrated to build a range of target applications, including traditional systems and most modern applications such as web services in a service-oriented architecture.

However, in many cases mismatches between pre-qualified available components and the specific reuse context of particular applications are inevitable and have been a major hurdle of wider component reusability and smooth composition. Component adaptation has been researched over the years as a key solution to the above problem [3][9][10][17]. 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 as a result of lack of automation in their adaptation process [9][17].

In this paper, a Generative Aspect-oriented component adaptatIoN (GAIN) approach is proposed to achieve adaptation at deeper level and meanwhile with high automation

of the process. The approach is based on the successful points in a few technologies, i.e., Aspect Oriented Programming [11], Software Product Line [1][5] and Generative Component Adaptation [2][4]. In GAIN approach, component adaptation is carried out within an aspect-oriented component reuse framework by generating and then applying the adaptation aspects under designed weaving process according to specific adaptation requirements. The generation absorbs the variation concept of software product line and assures the perfect suitability of adaptation aspects for the specific adaptation requirements of aimed reuse context. Compared with traditional AOP, the weaving process of aspects in GAIN supports more complex control flow, i.e., not only sequence, but also switches, synchronization and multiple threads, to make the adaptation more accurate and efficient for components reused in more complicated environments such as concurrent dynamic applications. To facilitate the reusability of adaptation knowledge, an expandable library of reusable adaptation aspects at multiple abstraction levels has been developed. A prototype tool is developed to scale up the approach.

The reminder of the paper is organized as follows: Section 2 discusses the related works with critical analysis. Section 3 describes the approach framework. Section 4 presents how to generate and apply reusable adaptation aspects under the designed weaving process. Section 5 introduces the prototype tool, and section 6 presents an example to demonstrate the approach. Finally, section 7 presents the conclusion.

2. Related works

2.1 SAGA project

Scenario-based dynamic component Adaptation and GenerAtion (SAGA) [9][17] at Napier University 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 on mainly 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 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 have to be developed to support the process, and probably the application domains has to be restricted as well.

2.2 Binary Component Adaptation

Binary Component Adaptation (BCA) [10] 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.

2.3 Superimposition

Superimposition [3] is a novel black-box adaptation technique, which is proposed by J. Bosch at University of Karlskrona/Ronneby. In Superimposition, 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, which degrades system efficiency.

2.4 Customizable Components

Customizable Components [12], as part of COMPOSE project, is an environment for building customizable software components, it is an approach to expressing customization properties of components. The declarations enable the developer to focus on what to customize in a component, as opposed to how to customize it. Customization transformations are automatically determined by compiling both the declarations and the component code; this process produces a customizable component. Such a component is then ready to be custom-fitted to any application.

In this work, the customized components generated for various usage contexts have exhibited performance comparable to, or better than manually customized code, however, component adaptation is limited to pre-defined optional customization, and deeper adaptation is not supported.

2.5 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 run-time.

It is distinguished between components that enhance and cross-cut 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.

2.6 JAsCo

JAsCo [15][16] is an aspect based research project for component based development, in particular, the Java Beans component model. JAsCo combines the expressive power of AspectJ [6][11] 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. Hooks are generic and reusable entities and can be considered as a combination of an abstract pointcut and advice [15][16]. Because aspect beans are described independently from a specific context, they 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 means it can not be used in a heterogeneous system including different programming language implementations.

2.7 Conclusion

Due to the complex nature of the mismatch between reuse requirements and components, available component adaptation approaches are either only capable for adaptation at simple levels such as wrappers, or inefficient to use as a result of lack of automation in their adaptation process. Deep level component adaptation can be achieved through AOP.

Some AOP based frameworks have been developed to achieve reusable aspects. 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 [13]. Furthermore, current AOP techniques only support weaving aspects sequentially. 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.

3. The approach framework

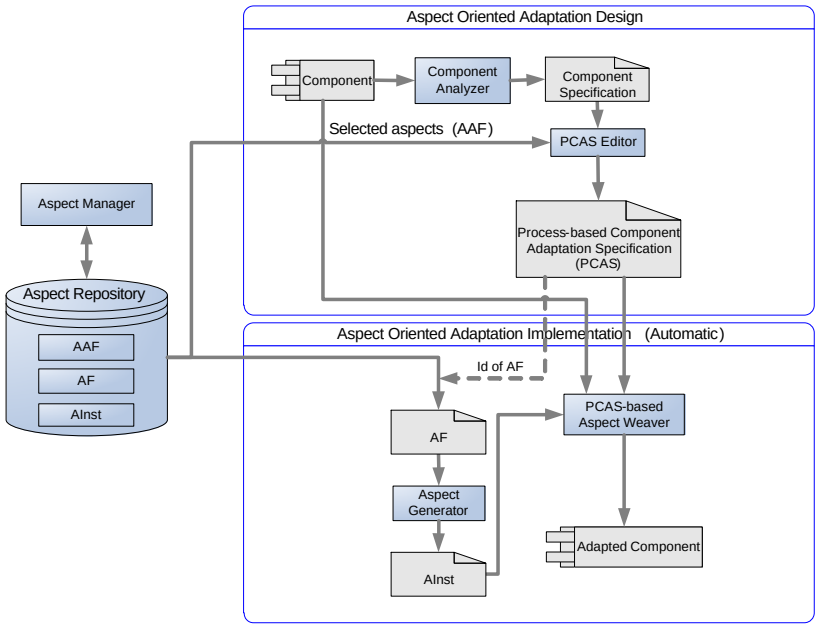

Fig. 1. The Generative Aspect-oriented component adaptatIoN (GAIN) framework

The general process of our approach is given in figure 1. We presume that a component has been found potential suitable to be used in a component-based application,

however, the application developer indicated some mismatches of the component and wishes to have it adapted.

The mismatches 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 composed by selecting aspects defined at the abstraction level of Abstract Aspect Frames (AAF). The selection of aspects is actually the process to determine functional variation of a specific adaptation. An AAF is considered as a template to coin out specific aspects. 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 result, platform variation is achieved during aspect generation. The input for the aspect generator is AF and the output is AInsts.

The aspect repository supports highly and incrementally reusable aspects. Reusable aspects are defined at different 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.

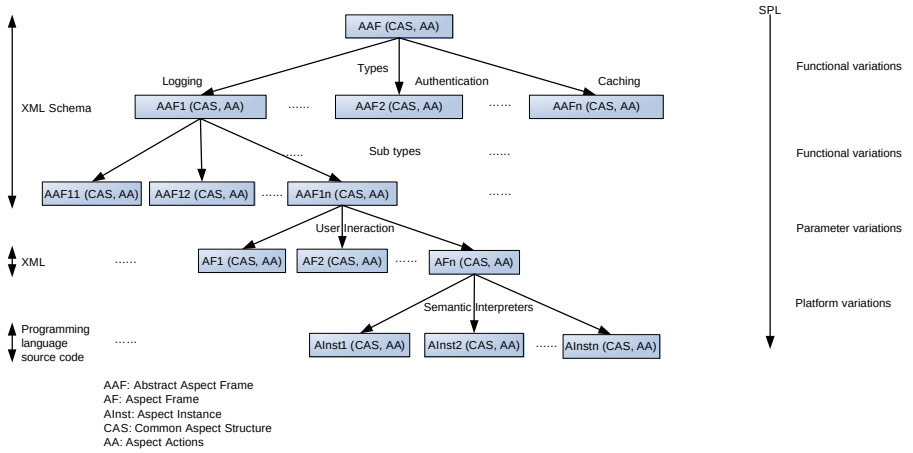

Fig. 2. Multiple abstraction levels of reusable aspects in Software Product Line view

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 the 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. Aspect-oriented generative adaptation

4.1 Capturing adaptation knowledge in aspects

In our approach, the adaptation knowledge is captured in aspects and aims to be reusable in various adaptation situations. As shown in figure 2, to achieve automated and precise adaptation, these aspects are defined at three abstraction levels, i.e., Abstract Aspect Frames (AAF), Aspect Frames (AF), and Aspect Instances (AInsts).

Abstract Aspect Frames are the fundamental and the most abstract level of the Aspect Repository. As XML schema files, AAFs are used to define the structure of different aspects. According to the functionality, the AAFs form 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.

AAFs are a hierarchical aspect type system defined in XML schema format. This type hierarchy includes many levels of aspect types and sub-types, which capture various functionalities of the adaptation aspects.

Each AAF may have many Aspect Frames. AFs are the second abstraction layer in aspect definition. AFs are the instances of related AAFs. Compared with its AAF, an AF has the details of a concrete aspect populated into it by assigning a value to the parameters. User interaction is required to create an AF from an AAF. Defined in XML format, AFs are independent from concrete AOP platforms such as AspectJ.

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 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 Aspect Instances from their AF is called Semantic Interpreter. The generation process is fully automatic.

The three abstraction levels of aspects facilitate the reusability of adaptation aspects as they realize different variations of these aspects, including functional variations, parameter variations and platform variations. At each level, a pair, namely (CAS, AA) is used to describe Common Aspect Structure (CAS) and Aspect Actions (AA). Common core assets are defined in Common Aspect Structures and variations are defined in Aspect Actions.

CAS provides the basic information of an aspect, e.g. which component to be adapted (target component), pointcut name, etc. All aspects have the same CAS at AAF level no matter how different these aspects are in functionality and implementation platform.

On the other hand, Aspect Actions provides the information of the variations of different aspects of the same or different aspect types. For instance, for an aspect of logging type, an output file name must be provided; similarly an authentication aspect must be supplied with an authentication type.

4.2 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.

The elements in a PCAS include target component(s) (“Host”), information of aspect(s) to be applied such as aspect id, type, and level (“aspect_level”), and process control information, such as execution mode (“Sequence”, “Switch”, and “Case”) and the condition. Since multiple aspects might access same resource such as a file, synchronization is supported in PCAS. A sample of PCAS structure is given in figure 3 with the data detail omitted, and a full example of the definition is given in section 6.

The elements in a PCAS include target component(s) (“Host”), information of aspect(s) to be applied such as aspect id, type, and level (“Apply-aspect”), and process control information, such as flow controls (“Sequence”, “Switch”, “Case”), conditions, and synchronization support (“synchronized”). Flow control elements are used to provide advanced weaving process, and synchronization support enables multiple accesses to the same resource such as a file or a database from different aspects. A sample of PCAS structure is given in figure 3 with the data detail omitted, and a full example of the definition is given in section 6.

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.

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

```
                comment="xxxx"/>
            </case>
        </Switch>
    </Sequence>
</Container>
</Host>
</AOP-Process>
```

Fig. 3. A sample of Process-based Component Adaptation Specification structure

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 CAS and AA. 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) *Semantic Interpreters*, which translate AFs to AInsts based on selected specific AOP platform and aspects. If there are m different AOP platforms and n different aspects in the tool, there will be m×n different interpreters. 4) *Component Analyzer*, which analyzes component and gets necessary information such as the class names and method names, for component adaptation. 5) *Aspect Generator*: based on AFs and corresponding Semantic Interpreters, executable aspect instances will be generated by Aspect Generator. The result executable aspects will be saved into aspect repository as AInsts. 6) *Aspect Weaver*, which is used to generate new components by weaving generated AInsts into original components.

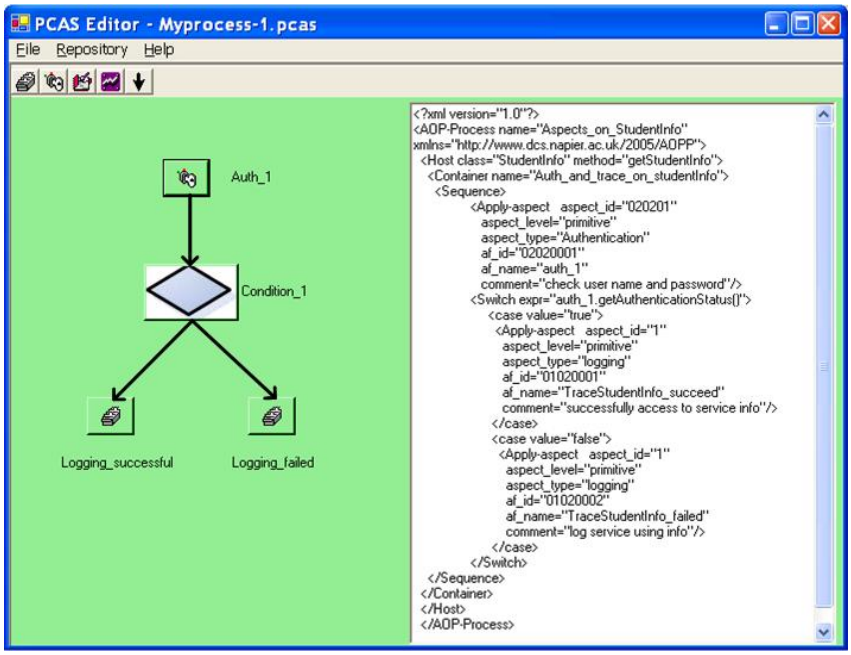

Fig. 4. A screen dump of PCAS Editor

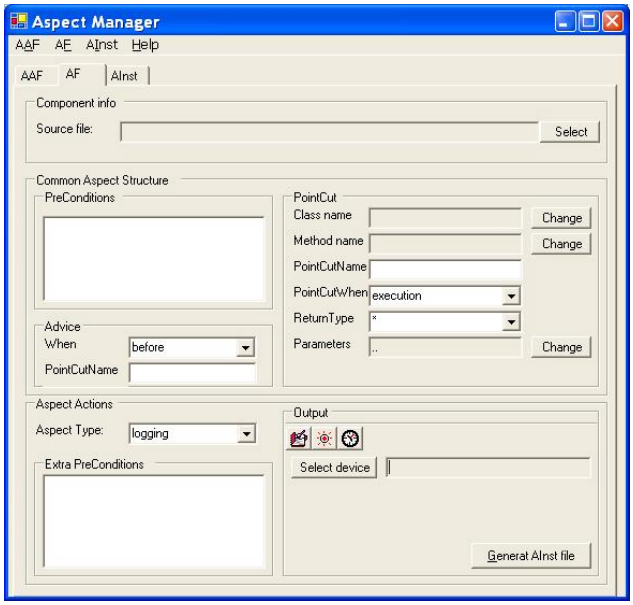

Fig. 5. A screen dump of Aspect Manager

6. Example

In this section, an intuitive example is given to demonstrate the proposed approach. We presume there is a component providing access to student information. A component user has found the component is potentially suitable for their business application and wishes to integrate the component into their system. However, the component user wants to restrict the access to the student information only to the approved users, and wishes to monitor the access by logging the usage time.

To respond to the above need, the component user plans to add authentication to this component prior to using it. According to the result of authentication, the detail of access activity to the component will be recorded.

An authentication aspect is applied to this component first, followed by the application of corresponding logging aspects depending on the result of authentication aspect.

The adaptation actions are then described in a Process-based Component Adaptation Specification (PCAS) shown in figure 6. As shown in figure 4, 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.

```
<?xml version="1.0"?>
<AOP-Process name="Aspects_on_StudentInfo"
xmlns="http://www.dcs.napier.ac.uk/2005/PCAS">
  <Host class="StudentInfo" method="getStudentInfo">
    <Container name="Auth_and_trace_on_studentInfo">
      <Sequence>
          <Apply-aspect   aspect_id="020201"
             aspect_level="primitive"
             aspect_type="Authentication"
             af_id="02020001"
             af_name="auth_1"
             synchronized="false"
             comment="check user name and password"/>
          <Switch
expr="auth_1.getAuthenticationStatus()">
              <case value="true">
                <Apply-aspect   aspect_id="1"
                  aspect_level="primitive"
                  aspect_type="logging"
                  af_id="01020001"
                  af_name="TraceStudentInfo_succeed"
                  synchronized="true"
                  comment="successfully access to ser-
vice info"/>
               </case>
               <case value="false">
```

```
                <Apply-aspect   aspect_id="1"
                  aspect_level="primitive"
                  aspect_type="logging"
                  af_id="01020002"
                  af_name="TraceStudentInfo_failed"
                  synchronized="true"
                  comment="log service using info"/>
              </case>
        </Switch>
    </Sequence>
  </Container>
  </Host>
  </AOP-Process>
```

Fig. 6. The Process-based Component Adaptation Specification

The specification in PCAS is at a rather overview level and does not contain the details of individual aspects. Developers need to provide parameter value for each aspect. Common AFs can be saved into Aspect Repository for further reuse. In this example, three AFs will be generated: AF for authentication, AF for logging if authenticated successfully, and AF for logging if authenticated unsuccessfully. Due to the structural similarity of AFs of different aspects, we only give the AF for logging if authenticated successfully in figure 7 as an example.

```
<?xml version="1.0" ?>
<Aspect name="TraceStudentInfo_succeed">
 <!-- Core asset -->
 <CommonCoreAsset>
  <PreConditions>
   <Precondition>java.io.*;</Precondition>
   <Precondition>java.util.*;</Precondition>
   <Precondition>org.aspectj.lang.*;</Precondition>
  </PreConditions>
  <PointCut>
    <Name>traceMethods</Name>
    <When>execution</When>
    <ReturnType>*</ReturnType>
    <ClassName>StudentInfo</ClassName>
    <MethodName>getStudentInfo</MethodName>
    <Parameters>..</Parameters>
  </PointCut>
  <Advice>
    <When>before</When>
    <PointCutName ref="traceMethods" />
  </Advice>
 </CommonCoreAsset>

 <!-- Variations -->
 <Variation type="logging">
  <ExtraPreConditions>
   <ExtraPrecondition/>
  </ExtraPreConditions>
  <Output>
   <Device>
    <File>D:\tmp\student_info_aop.log</File>
   </Device>
   <Messages>
```

CAS

AA

```
        <Message>Access to StudentInfo.getStudentInfo
                successfully on </Message>
        <Date/>
        <Message>at </Message>                                          → AA
        <Time/>
      </Messages>
    </Output>
  </Variation>
</Aspect>
```

Fig. 7. An Aspect Frame

From AFs, Aspect Generator generates aspect instances (AInsts) that are specific to a selected AOP platform. The generated AInst of the AF in figure 7 is given in figure 8.

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

public aspect TraceStudentInfo_succeed                                  → CAS
{
  pointcut traceMethods():execution(* StudentInfo.getStudentInfo(..));
  before():traceMethods() {

    Calendar cal = Calendar.getInstance();
    try {
      FileWriter   fw  = new FileWriter("D:\\tmp\\student_info_aop.log", true);
      PrintWriter pw = new PrintWriter(fw);

      pw.print("Access to StudentInfo.getStudentInfo
                successfully on " );
      pw.print(cal.get(Calendar.YEAR) + "." );
      pw.print(cal.get(Calendar.MONTH) + "." );
      pw.print(cal.get(Calendar.DAY_OF_MONTH) + " , ");                 → AA

      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 occured: " + e);
  }

  }                                                                     → CAS
}
```

Fig. 8. A simple Aspect Instance

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 new application is built and released to the targeted user.

7. 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 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 reuse framework: 1) the generation of adaptation aspects based on specific adaptation requirements and selected abstract aspects as template; 2) the advanced aspect weaving process definition mechanism that supports switch and synchronization; 3) an expandable library of reusable adaptation aspects at multiple abstraction levels.

The GAIN technology enables application developers to adapt the pre-qualified components to eliminate mismatches to the integration requirement of specific applications. The benefits of the approach include deeper adaptability, higher automation and therefore smooth component composition and wider reusability. As consequence, the target component-based systems will have better quality. Our case studies have shown that the approach and tool are promising in their ability and capability to solve the mismatch problem.

8. References

[1] Batory, D., "Product-Line Architectures", *Invited Presentation*, Smalltalk & Java in Industry and practical Training, Erfurt, Germany, October 1998.

[2] Batory, D., "Compositional validation and subjectivity and GenVoca generators", *Proceedings of the Fourth International Conference on Software Reuse*, 1996, pp. 166-175.

[3] Bosch, J., "Superimposition: a component adaptation technique", *Information and Software Technology*, 1999, 41, 5 pp. 257-273.

[4] Cleaveland, J. C., "Building application generators", *IEEE Software*, July 1998, pp. 5(4):25-33.

[5] *Diaz-Herrera, J.L., Knauber, P., & Succi, G.* "Issues and Models in Software Product Lines," *International Journal on Software Engineering and Knowledge Engineering*, 2000, 10(4):527-539

[6] http://www.eclipse.org/aspectj/

[7] http://www.sei.cmu.edu/cbs/

[8] Lieberherr, K., Lorenz, D., & Mezini, M, "Programming with Aspectual Components", *Technical Report NU-CCS-99-01*, March, 1999.

[9] Liu X., Wang B., & Kerridge J, "Achieving Seamless Component Composition Through Scenario-Based Deep Adaptation And Generation", *Journal of Science of Computer Pro-*

gramming (Elsevier), Special Issue on New Software Composition Concepts, 2005, pp. 56, 2.

[10] Keller, R., & Hölzle, U. "Binary Component Adaptation". *Proceedings of the 12th European Conference on Object-Oriented Programming*, July, 1998.

[11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W., "Getting Started with AspectJ", *Communications of the ACM*, October 2001, pp. 59-65.

[12] Kucuk, B., & Alpdemir, M.N., "Customizable adapters for blackbox components", *Proceedings of the 3rd International Workshop on Component Oriented Programming*, 1998, pp.53-59.

[13] Mehner, K., & Rashid A, "Towards a Generic Model for AOP (GEMA)", *Technical Report, No. CSEG/1/03. Computing Department, Lancaster University, UK*.

[14] Samentinger, J., *Software Engineering with Reusable Components*, Springer Verlag, 1997.

[15] Suvee, D., Vanderperren, W., & Jonckers V., "JAsCo: an Aspect-Oriented approach tailored for component Based Software Development", *Proceedings of the 2nd international conference on Aspect-oriented software development*, Boston, USA, 2003, pp. 21-29.

[16] Vanderperren, W., Suvée, D., Verheecke, B., Cibrán, M.A., & Jonckers, V, "Adaptive programming in JAsCo", *Proceedings of the 4th international conference on Aspect-oriented software development*, March, 2005.

[17] Wang, B., Liu, X., & Kerridge, J., "Scenario-based Generative Component Adaptation in .NET Framework", *Proceedings of the IEEE International Conference on Information Reuse and Integration*, Las Vegas, USA, November, 2004.