THAOP: An Aspect Oriented Programming Framework

Gang Feng[1], Qingxuan Yin[1], Xiaoge Wang[1]
[1]*Department of Computer Science and Technology, Tsinghua University, China*
feng_gang00@mails.tsinghua.edu.cn

Abstract

Aspect Oriented Programming (AOP) is one of the hottest research areas in the development of information technology during these years. People have found many programming problems for which the traditional Object Oriented Programming (OOP) model and Component Oriented Programming (COP) model are not sufficient to clearly capture some of the important design decisions the program must implement. This problem is much more obvious in Pervasive Computing Environment due to the variety of environment. AOP technology has been presented to resolve this problem. In this paper, we introduce THAOP, a lightweight and flexible Aspect Oriented Programming Framework based on TH_CORE (a.k.a. EFL) component platform, and compare THAOP to other AOP framework. And also a case study is given out to help readers have a much clearer visualize about THAOP.

Keywords: AOP, Component, Middleware, Pervasive Computing.

1. Introduction

The traditional Object Oriented Programming (OOP) model and Component Oriented Programming (COP) model have been presented as a technology that can fundamentally aid software engineering, because the underlying object model provides a better fit with real domain problems. But for years people have found many programming problems for which the OOP technology and the COP technology are not sufficient to clearly capture some of the important design decisions the program must implement, such as non-functional requirement. This problem actually results in "tangled" code which is really difficult to develop and maintain. To resolve this problem, Aspect Oriented Programming (AOP) [1] [2] technology has been presented. We propose THAOP, a lightweight AOP framework that could be used in Pervasive Computing Environment in this paper.

As an implementation of Component-Oriented Programming (COP) technique of software, TH_CORE has been focused on how to improve software quality attributes such as modularity, composability and reusability [3] [4]. It provides a TH_CORE component specification, a TH_CORE component runtime environment, as well as a series of assistant tools, including a compiler of IDL called EFLIDL. It is a lightweight middleware that can run well in Pervasive Computing Environment.

Though COP is considered to be a great software design and implementation technique, it is not mature enough to solve most problems in a highly flexible way. A lot of people has pointed out that there are certain kinds of software requirements, such as non-functional requirements, that must be implemented cross-cutting multiple classes, largely losing the modularity in object-oriented design and implementation code [5] [6] [7]. Therefore, it is not easy that components are reused without consideration of their low-level implementation details. AOP technology is an approach designed to handle complexities arising from such cross-cutting issues.

In THAOP, functional requirements are still encapsulated in each component, and particular non-functional requirements are flexibly tuned separately in the course of software composition. Code related to cross-cutting issues, called aspect, can be written in a way that need not align with the basic component structure. Actually, in THAOP aspects are also encapsulated in components which will be executed dynamically at certain position during the runtime of the application. To support the modularity for non-functional requirements in component-oriented software systems, we devise Aspect Definition Language Specification (ADLS) and Interceptor Base Interface Definition (IBID). We also extend EFLIDL, the compiler of IDL, to support the automation of AOP component development. With the help of THAOP, non-functional requirements such as synchronization, performance, physical distribution, fault tolerance, and so on, are enabled to be dealt with at component level. In another word, THAOP supports the modularity and manageability of non-functional requirements.

Related work is introduced in Section 2. Design and implementation of THAOP is described in Section 3.

This work was supported by the National High-Tech Research and Development Plan of China (No.2003AA1Z2090).

Then a case study and comparison to other AOP framework are given in Section 4 and Section 5. Finally, some conclusions are drawn from the presented work and some suggestions are made for future work.

2. Related Work

Aspect Oriented Programming (AOP) is one of the hottest research areas in the development of software technology during these years, so there is a broad range of research related to AOP. A large number of current applications of aspect oriented programming to middleware architectures focus on providing better modularization and support for QoS properties, or, more widely speaking, support for non-functional requirements in general. AspectJ [8] is one of the most famous AOP framework based on Java programming language. It uses static weaving technology to rebuild the aspect-oriented program to produce final program at compile-time. It provides powerful syntax extensions (i.e. aspects, advice and joinpoints) to capture crosscutting concerns that allow the programmer to perform global modifications on object sets in a very straightforward way. The JBoss application server [9] provides interceptors to allow the hosted applications to handle crosscutting concerns and supports common crosscutting concerns in J2EE application servers such as persistence, security, and transaction. The JAC (Java Aspect Components) project [10] provides a framework to build aspect-oriented distributed applications in Java. The AspectIX project [11] provides a middleware that provides aspects as a generic mechanism to implement and control non-functional properties of a distributed object (e.g., properties related to the communication mechanisms used for a distributed object or the consistency policies between replicated data parts of the distributed objects). Aspect configuration can be controlled and changed at run-time with an immediate effect on the distributed object. The COMQUAD [12] project uses an approach to addresses non-functional aspects in component-based systems. CQML+ is an XML-based language in COMQUAD for the description of "quality characteristics". This description is then interpreted and maintained by the component containers. Colyer and Clement [13] show how to apply aspect orientation techniques in an industrial setting to refactor a major crosscutting concern from an application server. Spring AOP [14] uses IoC (Inversion of Control) mechanism to implement AOP. It provides a non-invasive and lightweight AOP framework for Java programming language. Jiazzi [15] enhances Java with separately compiled, externally linked code modules called units. Units can act as effective "aspect" constructs with the ability to separate crosscutting concern code in a non-invasive and safe way. Besides, on Microsoft .NET platform, projects are also carried out to provide AOP framework in .NET environment, such as Aspect Sharp [16], AspectDNG [17] and Eos AOP [18].

Our research differs from many of the above approaches by focusing on the convenience, flexibility and lightweight of THAOP, and an approach to improve the customizability of the middleware platform itself. And the mainstream of middleware implementations is "*heavyweight, monolithic and inflexible*" [19]. That is what we want to avoid in our TH_CORE platform. Some parts of the platform could be integrated by THAOP and so can be deserted from the platform core. This improves the customizability of the middleware platform. With these advantages, TH_CORE and THAOP could be used in pervasive computing environment.

3. Design and Implementation of THAOP

THAOP is mainly consisted of AOP Library (AOPL), Aspect Definition Language Specification (ADLS), Interceptor Base Interface Definition (IBID) and Extended EFLIDL (E-EFLIDL), the AOP-supported IDL compiler. In order to help readers to have a clear understanding about THAOP, we first explain some AOP terms in our THAOP environment.

3.1 Terms Explanation

Join point: A join point represents a well-defined point in the execution flow of the component program at which the interceptor code can be executed. THAOP is based on TH_CORE, a component platform, on which components are treated as the basic program unit. So we define three kinds of join points: before, after and in place of a component method invoking.

Pointcut: A pointcut construct denotes a collection of join points. For example, invokinglog, a pointcut construct, picks out each join point of "before" kind in the execution flow of the component program. This pointcut can be used to create log file which helps developers and users to understand the execution flow of the running program.

Advice: THAOP interceptor code can be executed before, after or in place of the program execution when a join point is reached. These actions are defined using THAOP specific constructs before, after, and around. These constructs are called advices.

Aspect: An aspect module in THAOP contains pointcuts and the associated advices. Examples of aspects, often simply referred to as a system's cross-cutting concerns, include security, reliability, manageability, and, further, non-functional requirements. In THAOP, we use C or C++ to develop the components and also the advices. Actually advices in THAOP are TH_CORE components that implement IInterceptor interface, called Interceptor, which is introduced as the following.

Interceptor: In THAOP, interceptors stand for TH_CORE components that implement the IInterceptor interface. These components are used to modularize crosscutting concerns. THAOP executes interceptor code at certain join point in the execution flow according to the related aspect description. Interceptor encapsulates advice as component, greatly improving the modularization of cross-cutting concerns. To make the development of interceptors simpler for most developers, we define an interface named IInterceptor as the base interface of all interceptors. Developers could simply inherit this interface to build their particular interceptors. And developers need not implement all methods of this interface. They only need to implement methods which they are interested in.

3.2 The AOP Library (AOPL)

THAOP provides a library to support Application Program Interface (API) of AOP, named AOP Library (AOPL). AOPL provides the AOP support, simply referred to AOPL Interface, AOPL Runtime and dynamic aspect weaver. The architecture of AOPL is shown in Figure 1.

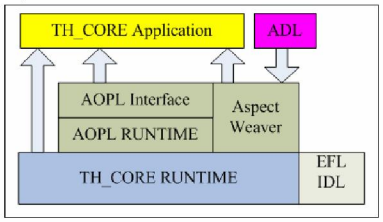

Figure 1: the architecture of AOPL

AOPL mainly provides two methods for clients to invoke as AOPL interface, as shown in Figure 2. Developers can easily enable or disable the AOP support to their TH_CORE application by invoking these two methods or not.

```
/* to initial the aop library */
int init_aop();
/* to finalize the aop library */
void finalize_aop(int);
```

Figure 2: Main AOPL interface

TH_CORE component platform has been focused on providing a flexible and efficient component platform. AOPL continues the effort on efficiency. The C programming language is the only language used in implementation of AOPL. Hash table is used to store aspect constructs in the memory. As dynamic weaving technology is applied, aspects information is read during the runtime of the TH_CORE application. Using hash table reduces the cost of searching aspects, greatly improving the efficiency of AOPL.

Dynamic aspect weaver weaves aspects with main TH_CORE program during runtime according to the ADL file related to the application, as described in Section 3.3. In THAOP, the dynamic aspect weaver is integrated into AOPL. The user can simply modify the ADL file to add, delete or modify aspects definition. The modification is effective once the application restarts.

3.3 Aspect Definition Language Specification (ADLS)

THAOP uses dynamic weaving technology to weave aspects and components. To support this, we propose Aspect Definition Language Specification (ADLS). The weaving compiler inside the AOP Library weaves components with aspects according to the aspect definition that is described by Aspect Definition Language (ADL). Final system is never built until runtime. The user can simply modify the ADL file related to the program if he (she) wants to redefine aspects. Restarting the program is a requirement to make the new definition effective.

ADL is a XML based language for aspect definition. We devise the specification for ADL, including the XML DTD for Aspect Definition Language, as shown in Figure 3, and several related rules that ensure the semantic correctness in the definition of aspects.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT aspects (aspect*)>
<!ELEMENT aspect (bind+, advice+)>
<!ELEMENT bind EMPTY>
<!ELEMENT advice EMPTY>
<!ATTLIST bind
        bind_pointcut    CDATA #REQUIRED
        bind_component   CDATA #REQUIRED
        bind_interface   CDATA #REQUIRED
        bind_method      CDATA #REQUIRED
>
<!ATTLIST advice
        advice_component CDATA #REQUIRED
        advice_interface CDATA #REQUIRED
>
```

Figure 3: XML DTD for ADL

To give readers an intuitionistic understanding on the ADLS, we provide an example of aspects definition that is consistent with ADLS, as shown in Figure 8.

This is a very simple example as in the whole file only one aspect construct is defined. In fact, according to the XML DTD given in Figure 3, there may be one or more aspect constructs in the same file. Each aspect construct may contain one or more bind constructs and one or more advice constructs. Bind constructs stand for pointcuts in THAOP, while advice constructs stand for advices, which are corresponding with interceptors. Aspect construct without bind constructs or that without advice constructs is not allowed in ADLS.

In order to clearly express the exclusive signification in semantics that an ADL file wants to show, several rules are made as the following.

Rule 1: A pointcut is allowed to be associated with one or more advices. This leads up to that in ADL a bind construct is allowed to be associated with multiple advice constructs. Advice constructs that are connected to one pointcut could either be contained by one aspect construct, or be separated in several different aspect constructs.

Rule 2: A bind construct is associated with all advice constructs that are contained by the same aspect

construct. That is to say, in one certain aspect construct, all bind constructs are associated with all advice constructs. For example, if an aspect element has two bind child elements, bind 1 and bind 2, and has two advice child elements, advice A and advice B, then bind 1 and bind 2 both are associated with advice A and advice B. Advice A and advice B will be executed in certain order when the program execution flow reaches any join point defined in bind 1 and bind 2 constructs. Bind constructs have nothing to do with advices that have different aspect father constructs.

Rule 3: Codes of all the advices related to a pointcut are executed when a join point of this pointcut is reached in the program execution flow, unless this pointcut is a "around" construct. They are executed in the order that they are defined in ADL.

Rule 4: For the pointcut of "around" construct, only one advice will be executed in the final program when the join point is reached. If there are two or more advices associated with the pointcut, the last one defined in ADL will be the final one.

3.4 Interceptor Base Interface Definition (IBID)

All Interceptors in THAOP implement the IInterceptor interface, called Interceptor Base Interface (IBI). We devise the Interceptor Base Interface Definition (IBID) here, as shown in Figure 4.

```
import "unknwn.idl";
[
  iid("Interface://www.efl.org/IInterceptor"),
  helpstring("Basic interface for Interceptor")
]
interface IInterceptor : IUnknown
{
    /* advice of before construct */
    HRESULT BeforeExecute(DISPPARAMS *pParams);
    /* advice of after construct */
    HRESULT AfterExecute(DISPPARAMS *pParams, HRESULT *hr);
    /* advice of around construct */
    HRESULT AroundExecute(DISPPARAMS *pParams);
    /* to initial the interceptor */
    void InitInterceptor();
}
```

Figure 4: Interceptor Base Interface Definition

There are three methods that mainly show the functionality of interceptors, BeforeExecute, AfterExecute and AroundExecute. This three methods stand for before, after and around advice constructs, being executed at certain join point in the execution flow of the final system.

BeforeExecute: This method is invoked at certain point cut with kind of "before". The parameter pParams of type DISPPARAMS encapsulates all of the arguments of the intercepted method from left to right. Developers could use this parameter to do works like checking arguments, modifying arguments and security certificate. The same parameter with the same functionality also appears in the following two methods.

AfterExecute: As the actual implementation of after advice construct, this method is invoked at certain point cut with kind of "after". It is a convention for TH_CORE components to have a return value of type HRESULT for all theirs' methods. Therefore the method AfterExecute has a parameter named hr of type HRESULT. That is to say this parameter stands for the return value of the intercepted method. Developers can check and even modify this value, but note that modifying this parameter may be dangerous and is not recommendatory.

AroundExecute: This method is invoked at certain point cut with kind of "around". When a join point of this kind is reached, the interceptor code will be executed in place of the program execution. In this situation, the original TH_CORE component method won't be executed. Note that users can register a number of advices of type around at the same point cut, but within all these advices only the last one can be executed. Other advices will be ignored by THAOP.

3.5 Extended EFLIDL (E-EFLIDL)

TH_CORE uses Interface Definition Languages (IDL) to define interface and class. An IDL compiler called EFLIDL is provided to compile IDL files into C or C++ programs. To support THAOP, we extend EFLIDL to make it able to produce AOP-supported components in a simple way. E-EFLIDL produces a component implementation program and a component proxy/stub program according to the IDL file. These programs are then compiled into executable binary codes that supports AOP, in another word, could be intercepted.

3.6 Scope of Interceptors

In THAOP, the effects of interceptors are strictly limited in the client processes that register these interceptors. Only the program using AOP feature may be affected by changing related aspect script and interceptors that are registered. Other programs upon the TH_CORE platform won't be tangled by unregistered interceptors. And of cause all interceptors won't affect the TH_CORE middleware platform itself unless some of the service processes of the platform have specified certain interceptors to be allowed to run in the process spaces of these services.

TH_CORE component platform supports Remote Procedure Call (RPC). In the runtime environment of TH_CORE, local client programs could use RPC to invoke methods that provided by other components in remote servers. In the RPC situation, the interceptor code will be executed locally in the client process space. The remote server process that provides the component RPC service won't be affected by any local interceptors. Actually, if a local client program wants to invoke a remote method, it instead invokes the same method of the proxy of the remote component. The proxy will deal with network communications backdoor. The interceptors intercept invoking of proxy methods and execute interceptor codes locally.

4. Case Study

In this section, we present a case study that uses THAOP to implement a temporary cross-cutting functionality. We have developed a program to resolve the shortest-path problem in several ways using TH_CORE component technology. ShortestPath, the main component used in this program, has three functional methods, as shown in Figure 5.

```
/* to calculate shortest path using dynamic programming */
HRESULT ShortestPathDP([in] struct PathInfo *pdata, [out] int *result);
/* to calculate shortest path using greedy algorithm */
HRESULT ShortestPathGreedy([in] struct PathInfo *pdata, [out] int *result);
/* to calculate shortest path using enumeration */
HRESULT ShortestPathEnum([in] struct PathInfo *pdata, [out] int *result);
```

Figure 5: Main methods of ShortestPath component

Now we want to analyze the cost of the dynamic programming method in different program contexts. As the program could be large and complex, it is not wise to modify the program to calculate the cost of every method call. We use THAOP to resolve this problem. The cost analysis of each method call, as the cross-cutting concern, is encapsulated into an interceptor, as shown in Figure 6.

```
import "interceptor.idl";
[
  iid("Interface://www.efl.org/ICostAnalyzer"),
  helpstring("Cost Analyzer which is a interceptor")
]
interface ICostAnalyzer : IInterceptor
{
}
[
  clsid("Class://www.efl.org/CostAnalyzer"),
  version(1.0),
  helpstring("Cost Analyzer class which is a interceptor")
]
class CostAnalyzer
{
   interface ICostAnalyzer;
}
```

Figure 6: Cost Analyzer Interceptor

We use the "around" construct to implement the cost analyzer, so we only need to implement the AroundExecute method of the interceptor, as shown in Figure 7.

```
HRESULT __stdcall _ICostAnalyzer_AroundExecute(
    ICostAnalyzer* iface,
    DISPPARAMS* pParams )
{
    assert ( pParams.cArgs >= 3 );
    VARIANT *arg = pParams.args;
    IPath *ip = (IPath *)(arg[0].value.PVOIDVal);
    struct PathInfo *pinfo =
        (struct PathInfo *)(arg[1].value.PVOIDVal);
    int *result = (int *)(arg[2].value.PINTVal);
    /* get the begin time here */
    ...
    /* call the original method */
    IPath_ShortestPathDP(ip, pinfo, result);
    /* get the end time here and output the cost*/
    ...
    return S_OK;
}
```

Figure 7: Implementation of AroundExecute

We register the interceptor, as shown in Figure 8.

Run the program and we then get the cost information of each method call of ShortestPathDP. This example could be extended to help to optimize programs. We can get the cost information of each method call in the program and find the bottleneck of the program, and then we can aim at the bottleneck to optimize the program.

```
<?xml version="1.0" encoding="UTF-8"?>
<aspects>
 <aspect>
  <bind
   bind_pointcut="around"
   bind_component="Class://www.efl.org/ShortestPath"
   bind_interface="Interface://www.efl.org/IPath"
   bind_method="ShortestPathDP"/>
  <advice
   advice_component="Class://www.efl.org/CostAnalyzer"
   advice_interface="Interface://www.efl.org/ICostAnalyzer"/>
 </aspect>
</aspects>
```

Figure 8: ADL file for CostAnalyzer interceptor

5. Comparison to Other AOP Framework

Many AOP Frameworks, such as AspectJ [8], AspectDNG [17] and Eos AOP [18], use static weaving technology to weave component program and aspects in the compile-time to produce final system. This technology may improve the efficiency of the program to a certain degree, but it brings new problems for the development of software. For example, the modification of programs in the final period of the development of software could bring in great effect to the final system. Compared to this, THAOP uses dynamic weaving technology, which weaves component program and aspects in the runtime. And unlike some of the AOP Frameworks such as JBOSS [9] and Spring AOP [14], THAOP is written in the C programming language. This results in that THAOP also has excellent efficiency although it uses the dynamic weaving technology.

Most AOP Frameworks in the current world are heavy-weight as there is always a middleware that the framework depends on. Although THAOP also depends on a middleware, the middleware, TH_CORE, is small and flexible enough to be able to run in the pervasive computing environment. So we say that THAOP is a light-weight AOP framework. And THAOP is much more flexible than many other AOP frameworks. Users can dynamically change the final system by simply modifying ADL file.

AspectJ and Eos AOP extend their component language, adding some constructs such as aspect, before and after to support AOP. This takes a little time for the users to learn the extended language. THAOP provides ADL, a simpler way, to describe aspects. ADL is based on XML, a widely used language in these days. Advices are encapsulated in a simple way as interceptors, which are also TH_CORE components.

6. Conclusions and Future Work

Like other middleware platforms, such as CORBA, DCOM, J2EE and .NET, TH_CORE provides abstraction, simplicity and uniformity for the development of distributed applications. But unlike other middleware platforms, TH_CORE is

customizable as a lightweight middleware platform. TH_CORE itself has architecture of good modularity and extensibility. With some modules deserted, TH_CORE could run well in Pervasive Computing Environment, which has rigorous resource limitation. Users can tailor the middleware to fit their specific needs.

The design and implementation issues of THAOP are presented and discussed in the paper. As we all know, the main goal of the integration framework for software composition is to support separation of concerns enough to deal with non-functional aspects and functional aspects, if possible, at an application-builder level. But OOP and COP techniques for developing complex systems do not support such advanced separation of concerns at the software architecture level. THAOP presented in the paper is able to promote modularity and understandability as well as allows incorporating nonfunctional aspects in a flexible way through modular composition. THAOP is different from most other AOP frameworks, having some useful advantages compared to other AOP frameworks.

There are several issues that we would like to investigate further, such as the security of THAOP, context awareness in interceptors and optimization of AOPL. Another important research topic is to make the TH_CORE platform, as well as THAOP, more suitable for the resource-limited environment. And we would try to use THAOP to implement some of the foundational middleware functionality such as persistency.

References

[1] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, "Aspect-Oriented Programming", *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland.* Springer-Verlag LNCS 1241. June 1997.

[2] Robert E. Filman, "What Is Aspect-Oriented Programming, Revisited". *Workshop on Advanced Separation of Concerns, 15th European Conference on Object-Oriented Programming*, Budapest, Jun. 2001.

[3] Yanni WU, Kuo ZHANG, Xiaoge WANG, Jinlan TIAN, "Extending Metadata with Scenarios in Adaptive Distributed System", *International Conference on Information Technology and Applications*, Volume2 pp. 63-68, 2005-07.

[4] Kuo ZHANG, Xiaoge WANG, Yu Chen, Yanni WU. "A Component-Based Reflective Middleware Approach to Context-Aware Adaptive Systems". *The Fifth International Conference on Web Engineering*, 2005-07. LNCS3579, pp.429-434, 2005.

[5] Charles Zhang and HansArno Jacobsen, "Resolving Feature Convolution in Middleware Systems", *OOPSLA'04*, Oct. 2428, 2004, Vancouver, British Columbia, Canada.

[6] Joon-Sang Lee, Doo-Hwan Bae, "An aspect-oriented framework for developing component-based software with the collaboration-based architectural style", *Information and Software Technology 46 (2004) 81–97.*

[7] Charles Zhang and Hans-Arno Jacobsen, Member, IEEE, "Refactoring Middleware with Aspects", *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, VOL. 14, NO. 11, NOVEMBER 2003.

[8] G. Kiczales et al., "An Overview of AspectJ," *ECOOP 2001 - Object-Oriented Programming*, LNCS 2072, Springer-Verlag, 2001, pp. 327–353.

[9] JBoss URL:http://www.jboss.org.

[10] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, Laurent Martelli, "JAC: An Aspect-Based Distributed Dynamic Framework", *Software: Practise and Experience (SPE)*, 34(12):1119--1148, October 2004.

[11] Franz J. Hauck, Ulrich Becker, Martin Geier, Erich Meier, Uwe Rastofer, Martin Steckermeier, "AspectIX: A Middleware for Aspect-Oriented Programming", *In Object-Oriented Technology, ECOOP'98 Workshop Reader*, LNCS 1543, Springer, 1998.

[12] Steffen Gobel, Christoph Pohl, Simone Rottger, and Steffen Zschaler. "The COMQUAD Component Model: Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects". *Proceedings of the 3rd International Conference on Aspect Oriented Software Development.* ACM, 2004.

[13] Adrian Colyer and Andrew Clement. "Large-scale AOSD for middleware". *In 3rd International Conference on Aspect-oriented Software Development (AOSD'04)*, pages 56 – 65, Lancaster, UK, 2004.

[14] "Spring AOP: Aspect Oriented Programming with Spring", available at http://www.springframework.org/docs/reference/aop.html.

[15] Sean McDirmid, Wilson C. Hsieh, "Aspect Oriented Programming with Jiazzi", *In Proceedings of the 2nd International Conference Aspect-Oriented Software Development*, pages 70--79, Boston, Massachusetts, Mar. 2003. ACM.

[16] Aspect Sharp User Preferences. Available at http://aspectsharp.sourceforge.net/.

[17] Available at http://sourceforge.net/projects/aspectdng/.

[18] Publications and Documentation about Eos are available at http://www.cs.virginia.edu/~eos.

[19] Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. "The design of a configurable and reconfigurable middleware platform". *Distributed Computing*, 15(2):109–126, 2002.