A Connector-Centric Approach to Aspect-Oriented Software Evolution

Yiming Lau, Wenyun Zhao, Xin Peng, Shan Tang
Software Engineering Lab, Computer Science and Engineering Department,
Fudan University, Shanghai 200433, China
{051021050, wyzhao, pengxin, tangshan}@fudan.edu.cn

Abstract

Lose sight of the existence of system crosscutting concerns, e.g. safety and quality etc, often causes the system hard to maintain and evolve according to the changing environment and requirements. In this paper we propose an incremental Aspect-Oriented (AO) approach to ease this kind of evolution problem in architecture level. In this approach we introduce a novel connector, namely Aspect Weaving Connector (AWC), to support the seamless integration of AOSD and software architecture modeling. Concretely crosscutting concerns are encapsulated into aspects and modeled as software components. AWC acts as a connector wrapper coordinating the interaction between aspectual and regular components. In order to provide a formal basic to AWC, we propose a conceptual model of it, which formalizes the underlying mechanisms of aspect dynamic weaving in architecture level using process algebra CSP. Then we verify the model's properties with FDR2 and prove that our connector-centric AO architecture modeling approach can give system an architectural dynamism and make it easier to maintain and evolve.

1. Introduction

Nowadays, software development is becoming increasingly complex. One reason is that the structure and behavior of today's software often need to be evolved throughout its life cycle to handle crosscutting concerns (e.g. *safety* and *quality* etc.) according to its changing environment and requirements. So we need to develop software systems in an easy maintenance way. The integration of Component-Based Software Development (CBSD) [8] and Aspect-Oriented Software Development (AOSD) [3], can provide such a way for us to develop today's evolving system.

In this work, we focus on the seamless integration of AOSD and Software Architecture (SA) modeling to support software evolution. Generally, the main effort made in the AOSD is at the implementation level, from which Aspect-Oriented Programming (AOP) has emerged as a new paradigm of software development and evolution. The concept and advantages of aspect, and how it is weaved with the kernel class by means of the *pointcuts*, *joinpoints* and *advice* methods (*after/before/around*) are clear at the implementation level. However, few works has been done concerning aspects at architectural level [2].

During architecture modeling, the aim of most architects is to define system functional components and their interrelationship. And they usually lose sight of the existence of system crosscutting concerns, e.g. safety and quality etc. Nevertheless, such negligence often causes the system hard to maintain and evolve, for crosscutting concerns often contaminating components and connectors and making SA extremely complex [1].

In this paper we propose an incremental AO architecture modeling approach to ease this kind of complexity. In this approach we introduce a novel connector, namely *Aspect Weaving Connector* (AWC), to seamlessly integrate AOSD and SA modeling. Concretely, system crosscutting concerns are encapsulated into aspects and modeled as components separately. AWC acts as a connector wrapper and specifies the coordination between aspectual and regular components. In order to provide a formal basic to AWC, we propose a conceptual model of it, and employ CSP [6] to formalize the underlying mechanisms of aspect dynamic weaving in architecture level. Then we use model checking tool FDR2 [7] to verify the properties of this model, and prove that our connector-centric AO architecture modeling approach can give system an alternative architectural dynamism and make its maintenance and evolution easier.

The article is organized as follows: in section 2, the proposal of AO architecture modeling is presented. And section 3 presents a concept model of AWC and section 4 shows how to formalize it using CSP. Then in section 5, we verify the model's properties by FDR2. Finally, related works, conclusions and future work are presented in section 6.

31st Annual International Computer Software and Applications Conference(COMPSAC 2007)
0-7695-2870-8/07 $25.00 © 2007 IEEE

2. A Proposal to Incremental AO Architecture Modeling

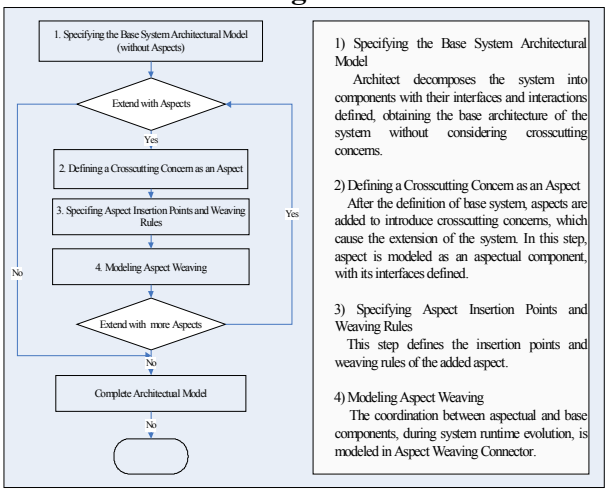

Fig. 1. Steps of the approach

The proposal presented here refers to an aspect-oriented and component-based incremental SA modeling approach. In architecture-designed phase, software system is firstly defined as a set of components. These components define system's high-level functionality and are named as *"base"* components. And architectural connectors are used to manage the binary interaction (or communication) between these components. This kind of *"base"* system hasn't taken crosscutting concerns into consideration. So after the definition of base system, the approach incrementally takes the crosscutting concerns that cross architectural components into consideration as aspects.

To seamlessly integrate the concepts of AOSD and SA modeling, aspect is defined as a special kind of component, namely "aspectual" component, which can be manipulated by AWC. Then system's evolution is obtained by such aspectual components' dynamic weaving, which is managed by AWC based on aspect *insertion points* (IPs) and *weaving rules* (WRs). In our AO architecture modeling, IPs specifies which component interface events will trigger the aspectual component's execution, which is close to the *join points* concept of AOP [10]. WRs is similar to *point cut designator (pcd),* which specify under what circumstances the aspectual component is to be weaved into system through IPs. The steps of this modeling approach are shown in Figure 1.

2.1. The Example

To clarify the above concepts, we introduce a typical e-commerce application *Java Pet Store* (JPS) to illustrate our approach.

"An online pet store sells consignation products, i.e. animals, to customers. The application has several user interfaces (e.g. WebClient, WAPClient, AppClient, etc.), receiving requests from customers and passing orders to the *order process server* (OPS).

Now, in order to ensure the security and integrity of the business, an approve operation is needed by third-party product supplier to ensuring final deliver service. This new added operation will generally affect some basic functions of the system, e.g. order operations."

From the requirement specification above, we begin JPS's AO software architecture modeling:

1) Firstly, we identify that the base architecture of JPS consists of two kinds of base components, Client and Server, which is represented in Fig.2a. Of course, the base architecture of JPS system can be described in an ADL, e.g. ACME [11].
2) Then, we define approve operation as a security concern of the system, and model it as an aspectual component, i.e. Approval.
3) According to the security rank of the system, Approval will be weaved into the base JPS system to perform approve operation, which constraint customer's order operation for the sake of the security of the trade. So, we identify that the IPs of the approval aspect are the events coming from Client components' order interfaces. And the WRs for Approval are:

*IF [*Client.order*] and [SecurityRank=Low] THEN DO[Approval.approve](after)[Server.accept];*
*IF [*Client.order*] and [SecurityRank=Med] THEN DO [Approval.approve](before)[Server.accept];*
*IF [*Client.order*] and [SecurityRank=Hi] THEN DO [Approval.approve](around)[Server.accept];*

The Approval's insertion points and weaving rules define the runtime evolution strategies of JPS.

4) As to model Approval's dynamic weaving, we define an aspect weaving connector, i.e. AWC. Such connector wraps the connections and manages the coordination between components, i.e. Clients, Server and Approval. Moreover, it determines if and when the aspect will be applied or not, according to the weaving rules (Fig. 2b).

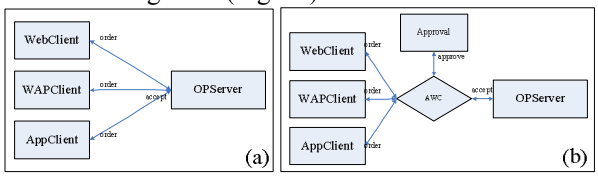

Fig. 2. (a) Base System (b) Extended System

3. A Conceptual Model of AWC

3.1. Analysis of Aspect Dynamic Weaving

In architecture level, dynamic aspect weaving can be seen as a software evolution mechanism causing

0-7695-2870-8/07 $25.00

aspectual component joins into system and interacts with base components without system interruption. Although the quality of service during weaving may decline a little, requests to base component should not be refused or cancelled and aspect weaving should be transparent to the clients. After aspect weaved into the system, it should have some kind of facility to coordinate the interaction between base and aspectual components. So aspect weaving at architecture level should include the following three main features:

1) **Coordinator:** It intercepts the coming requests from clients and then routes them to corresponding interfaces of base server components. If the aspect is weaved into the system, requests will be routed according to the weaving rules. If the *Adv* of the rule is *"before"*, the coordinator will firstly call weaved aspectual component's specific operation, then pass on the original request to the affected base server component. If the advice is *"after"*, the coordinator will pass on the request to the affected base component, then invoke weaved aspectual component's specific interface. Otherwise, if the advice is *"around"*, the coordinator will just call weaved aspectual component's specific operation instead of the affected component.
2) **Request Buffer:** During the process of aspect weaving, all the request events from insertion points are blocked and buffered. After aspect is weaved into system, the buffered requests are dumped and delivered to the aspectual and affected base component respectively according to the weaving advice (*before/after/around*). If the buffered requests have not been handled completely, new coming requests should also be buffered at this time.
3) **State Manager:** It manages the state between the affected base server components and weaved aspectual component. For instance, it initializes aspectual component's state as the base ones.

In the next sections, these three features will be formalized as a conceptual connector model. Then the behavior and properties of this model (i.e. *deadlock-freedom*, *client-transparency*, and *server-correctness*) will be verified.

3.2. Overview of Aspect Weaving Connector

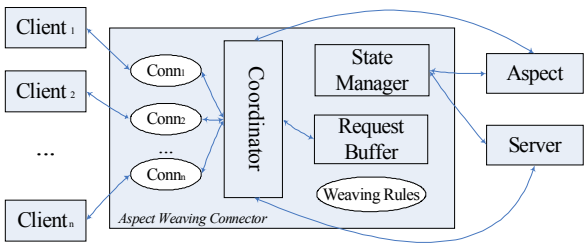

Fig. 3. Aspect Weaving Connector

In Figure 3, we introduce a conceptual connector model as the formal basic to AWC. It coordinates the interactions between aspectual and base components of the systems at runtime, e.g. clients, server and aspect. It contains several roles act as the aforementioned dynamic weaving features. The rules governing the weaving process are as follows:

1) Before aspectual component weaving into base system, requests will be intercepted by AWC and forwarded directly to base server component.
2) If aspect weaving begins, the requests should be serialized and stored into AWC buffer.
3) During weaving process, AWC synchronize the state between the aspectual and base components.
4) After weaving, AWC fetches out the buffered requests sequentially and calls aspectual components specific interface before (or, after/around) forward the request to the base server component, and then pass the result to the client components, according to the weaving rules. From then on, AWC coordinated the interaction between aspectual and base components. Finally, as the buffer emptied, all the new coming calls to the base component's specific interface will be routed as the buffered ones.

Based on the work [5] about connector formalization, we formalize AWC model in the following sections.

4. Aspect Weaving Connector Modeling

4.1. Process Notation

We employ process algebra CSP [6] to formalize AWC. CSP is a way to describe and analyze processes behavior patterns and interactions. We use an approach similar to Wright [5] to formalize connector behaviors.

```
Client = client.order→client.return→Client
Server = service.accept→service.result→Server
Glue = client.order→service.accept→Glue
     □service.result→client.return→Glue
BasicCSC = Client||Server||Glue
```

Fig. 4. A basic JPS client-server connector

To illustrate, Figure 4 shows a simple architectural connector BasicCSC. It is the parallel composition of three processes (Client, Server and Glue). Client and Server processes represent two roles of the JPS connector, standing for the behavior patterns of order interface of JPS Client component and accept interface of JPS Server component respectively. Client sends a request to order product from JPS and waits for a response repeatedly; then Server accepts the Client's order and returns results. client.order and client.return and so on are events used as the alphabet to describe processes. Glue process uses symbol □ to choose between two paths which are determined by environment (i.e. other process). The symbol || in CSP indicates processes synchronized on shared events,

which means an event shared in many processes can not occur until all processes are willing to engage in it. So process Glue coordinates the behaviors of Client and Server: a client.order event followed by a service.accept event, and server.result followed by client.return.

4.2. Modeling

To model the aspect dynamic weaving features, we need to modify BasicCSC by extending its Client, Server and Glue processes. It takes four steps:

1) Extend Client process. In the real world, there is usually more than one client that requests service concurrently. So we need to introduce more Client processes with index mechanism.
2) Extend Server process to model the new role of the aspectual component. To respond to different clients, Server also needs to be enhanced to deal with every incoming request respectively.
3) Extend Glue process to model aspect dynamic weaving features and synchronize the extended processes Client and Server.
4) Obtain AWC's behavior, i.e. AWC process.

```
Set ClientSet = {a,b,c}
Client(i) = client.order.i→client.return.i→Client(i)
Clients = |||i:C@Client(i)
```

Fig. 5. Three Client Processes

Figure 5 shows the new extended client process. It specifies a model with three concurrent client components marked by letters in the set ClientSet. The Clients process consists of three processes, Client(a), Client(b), and Client(c). Each of them sends requests and waits for returns independently.

```
Base = base.accept?i→base.result.i→Base
     □base.pause→base.getstate→base.continue→Base
     □base.setdelivermode?i→base.return.i→Base
Approval = aspect.acqstate→aspect.ready→Ready
     Ready = aspect.approve?i→
     aspect.changedelivermode!i→aspect.result.i→Ready
Weaver = donothing→Weaver
       ∏deweave→Weaver
       ∏weavebeforestart→base.pause→aspect.ready→
       base.continue→weavebeforeend→Weaver
      ∏weaveafterstart→…∏weavearoundstart→…
ExServer = Base||Aspect||Weaver\{donothing, deweave}
```

Fig. 6. The Extended Server Process

In Figure 6, we show the enhanced server process, which composed of three sub-processes: Base represents the server component (i.e. JPS Server), which provide service for client; Approval describes the behavior pattern of the aspectual component, which firstly initialize with base server component's state and then provide operation; Weaver is introduced to simulate the middleware which either does nothing or fires the weavestart event to weave the aspectual component into the system. As to whether to start an aspect weaving or not is determined by Weaver according to the weaving rules. Moreover, the variable i used in these processes is to identify each Client's request (base.accept?i) and respond (base.result.i) correspondingly.

To model aspect dynamic weaving features and synchronize the extended processes Client and Server, we introduce the following four intermediate process named BasicGlues, StateManager, QueueBuffer, and Coordinator to the Glue process.

```
BasicGlue(i) = client.order.i→service.accept!i→BasicGlue(i)
     □service.result.i→client.return.i→BasicGlue(i)
BasicGlues = |||i:C@BasicGlue(i)
```

Fig. 7. Basic Glues Process

For usually more than one client requests services, we need to introduce more BasicGlue process to correspond each Client processes. Then each BasicGlue is responsible for accepting request with the client's tag (client.order.i) and sending it to server (service.accept!i) and delivering responses from server (service.result.i) to correspondence client (client.return.i). Similar to Clients, the BasicGlues process in Figure 7 is composed of three interleaving BasicGlue processes.

```
StateManager = weavebeforestart→base.getstate→
     aspect.acqstate→weavebeforeend→StateManager
   □weaveafterstart→…□weavearoundstart→…
```

Fig. 8. State Manager Process

The process in Figure 8 plays the role of State Manager. In this case study, when aspect weave begin, it firstly gets the base server component's state, and then transfers it to the aspectual component by invoking interfaces (base.getstate and aspect.acqstate) provided by Base and Approval.

```
QueueBuffer = B(<>)
B(<>) = queuebuffer.left?x→B(<x>)
      □queuebuffer.empty→B(<>)
B(s) = if #s < N
     then  queuebuffer.left?x→B(s□<x>)
       □ queuebuffer.right!head(s)→B(tail(s)
     else  queuebuffer.right!head(s)→B(tail(s))
```

Fig. 9. FIFO Queue Buffer Process

The extended Glue still contains two key processes QueueBuffer and Coordinator. Figure 9 shows a process of FIFO buffer, namely QueueBuffer, with a maximal size of N, whose specification is adopted from [6]. And Coordinator can use it to buffer the incoming requests from different clients during aspect weaving.

For the complexity of Coordinator, we firstly analyse its behavior before and during aspect weaving in Figure 10. Before aspect weaving, Coordinator will try TryBase(i) process to call JPS server component's accept interface (base.accept!i) then try ReturnBase(i) process to wait for a server response (base.result.i), as client's order interface (client.order.i) sends requests to server through connector (service.accept?i). If the service provided

by base component invoked successfully, a response to Client(i) (indicated by tag i: base.result.i) is delivered to Coordinator, and the Return(i) process is activated. Then it will pass the response to process BaseGlue(i), which will deliver response to Client(i) accordingly.

```
Coordinator = (service.accept?i→TryBase(i);Coordinator)
      □(□i:C@ (ReturnBase (i); Return(i));Coordinator)
      □service.accept?i→(weavebeforestart□weaveafterstart
     □weavearoundstart)→serialize→queuebuffer.left!i→Buffering
Return(i) = service.result.i→SKIP
TryBase(i) = base.accept!i→SKIP
ReturnBase(i) = base.result.i→SKIP
TryAspect(i) = aspect.approve!i→aspect.changedelivermode?i→
            base.setdelivermode!i→base.return.i→SKIP
ReturnAspect(i) = aspect.result.i→SKIP
Buffering=service.modify?i→serialize→queuebuffer.left!i→Buffering
          □aspect.run→weavebeforeend→DumpWeaveBefore
          □aspect.run→weaveafterend→DumpWeaveAfter
          □aspect.run→weavearoundend→DumpWeaveAround
```

Fig. 10. Coordinator's behavior before and during Aspect Weaving

When weaving rules are satisfied and Weaver starts to weave the aspectual component into the system, the request events coming from insertion points (IPs) to base component interface (Com), will be serialized and stored in a buffer. Here, the IP is client.order.*. And Com is base.accept. Buffering process uses QueueBuffer to carry out the serialized task. As the aspectual component is ready, the buffered requests will be deserialized in order, and its behavior is specified in dump processes that will be discussed later.

```
DumpWeaveBefore = queuebuffer.right?i→deserialize→
     TryWeaveBefore (i); DumpWeaveBefore
     □service.accept?i→queuebuffer.left!i→DumpWeaveBefore□
     ((□i:C@ ReturnWeaveBefore (i)); DumpWeaveBefore(i))
     □queuebuffer.empty→WeaveBeforeRouter
TryWeaveBefore(i) = TryAspect(i); ReturnAspect(i); TryBase(i)
ReturnWeaveBefore (i) = ReturnBase(i);Return(i)
WeaveBeforeRouter =
   service.accept?i→TryWeaveBefore(i); WeaveBeforeRouter
   □((□i:C@ ReturnWeaveBefore (i)); WeaveBeforeRoter)
   □(service.accept?i→deWeave→TryBase(i); Coordinator)
```

Fig. 11. Coordinator's behavior for "before" Advice

The behavior of Coordinator after aspect weaving is divided into three parts for each kind of advice (*before*, *after* and *around*). Figure 11 describes the behavior of Coordinator after aspectual component weaved *before* base component. During aspect weaving the coming requests are buffered. As the aspectual component Approval that is weaved "*before"* the base server component is ready; Coordinator will firstly deserialize the buffered requests in order. And this behavior is specified in DumpWeaveBefore process. It indicates that if QueueBuffer is not empty, the buffered requests will be fetched out one by one and sent to the TryWeaveBefore(i) process. TryWeaveBefore(i) is a sequential composition of TryAspect(i), ReturnAspect(i) and TryBase(i). It will firstly try TryAspect(i), which invokes Approval's approve interface (aspect.approve!i), and coordinates the interaction between Approval and Server (aspect.changedelivermode?i->base.setdelivermode!i); then it try ReturnAspect(i) to delivers aspect's return (aspect.return.i). After that, the process tries TryBase(i) to invokes base server's accept interface (base.accept!i). If invoked successfully, DumpWeaveBefore will be ready to use process ReturnWeaveBefore(i) to return the result or to dump the subsequent buffered requests again. During the dumping process, new requests intercepted by Coordinator will be appended to the tail of the queue buffer. When the buffer becomes empty, the WeaveBeforeRouter process is activated. From then on, as requests coming (service.accept?i), this process will send them to TryWeaveBefore(i), then attempt to use ReturnWeaveBefore(i) to return result or receive new requests again. From then on, we regard the process of aspect weaving for *"before"* advice finished. Due to the space limit, we will not show the specification of Coordinator after aspectual component weaved *after* or *around* base server.

```
AWGlue = BasicGlue||Coordinator||QueueBuffer||StateManager
AWC = Clients|| ExServer || AWGlue
```

Fig. 12. Aspect Weaving Connector Process

In Figure 12, we have the extended glue AWGlue and the final aspect weaving connector process AWC. The model supports three client components accessing base server (JPS Server) concurrently. Of course, we can model more client components only by increasing the number of elements in set ClientSet.

5. Properties Checking

In this section, we employ FDR2 to check three properties of our AWC model, i.e. deadlock-freedom, client-transparency, and server-correctness.

First of all, using the build-in mechanisms of FDR2, we can conform that AWC is deadlock-free. As to client-transparency, it means that once a Client sends a request, soon or later, the Client will get a response, no matter whether the aspect weaved *before/after/around* the base Server or not. Moreover, the Client's interface needs not to be altered after aspect weaving. To checking client transparency, we use the "process-oriented specification" mechanism provided by FDR2 to describe our model and do the checking.

```
CT (i) = client.order.i→service.accept!i→service.result.i→
        client.return.i→CT(i)
SPEC_T = (|||i:C@ CT(i))|||CHAOS({|wrapper, weavestart,
            weaveend, base, aspect|})
assert SPEC_CT [T = AC
```

Fig. 13. Client Transparency

In Figure 13, we define process CT using Client and its corresponding BasicGlue processes' alphabet (client.order, service.accept, service.result, and client.return) to specify this property. CHAOS is a predefined process of FDR2 that means to ignore its following events when proving the property. The assert statement declare that our model should satisfy the specification process SPEC_CT, which can be proved by FDR2 automatically.

```
SC (i) = Before(i)□weavebegin→After(i)
Before(i) = client.order.i→(base.accept.i→Processing(i)→
            weavebegin→(aspect.approve.i→aspect.result.i
            □base.accept.i→base. result.i)→After(i))
Processing(i) = base.result.i→SC
              □weavebegin→base.result.i→After(i)
After(i) = client.order.i→(aspect.approve.i→aspect.result.i
            □base.accept.i→base. result.i)→After(i)
SPEC_SC =(|||i:C@SC(i))|||CHAOS({|service, weaveend,
    aspect.run, base.getstate, aspect.setstate, client.return|})
assert SPEC_SC [T= AWC
```

Fig. 14. Server Correctness

Server-correctness specifies that the request which is delivered to Server before aspect weaving (the weavebegin event) should be responded by the Base component, and the request delivered after the weavebegin event should be responded by the Aspect or Base component according to weaving rules. We use processes SC to model the possible event sequence according to the server correctness property shown in Figure 14. For clarity, we need to introduce two assistant processes Before and After which model the desired behavior of server correctness property before and after aspect weaving (Before □ weavebegin→After). The Processing process is a sub process of Before and handles the case that the weavebegin event occurs when the request is being processed. And these SC processes constitute the specification process SPEC_SC who ignores some events.

We use FDR2 to check the client transparency and server correctness properties. It will show a violation and event trace, if an illegal event sequence occurs. And FDR2 reports our model satisfied all the specifications.

6. Conclusions and Future Work

AO architecture modeling is also the aim of several works. In the PRISMA approach [9], aspects are model as new abstractions different form component and connector. F. Arbab [4] proposes that in software architecture, the aspects separation in a system can be treated as a coordination problem. Their ideas are similar to us, but our research focuses on providing a methodological approach to model AO software architecture, and seamlessly integrate AOSD and SA modeling to support software evolution by a novel kind of connector AWC. Then we put forward a conceptual model as the formal basic of AWC, which formalizes the mechanisms of aspect dynamic weaving in architecture level by using CSP. And then we verified its properties with FDR2, and proved that our connector-centric model has three main properties, i.e. *deadlock-freedom*, *client-transparency and server-correctness*. That is to say our connector-centric AO software modeling approach can make system maintenance and evolution easier, and give system an alternative architectural dynamism.

Our future work will be, on one hand, to obtain a more complete AO modeling framework, and on the other hand, to generalize and extend the existent ADL to obtain a new AO ADL.

7. Acknowledgments

This work is supported by the National High Technology Development 863 Program of China under Grant No.2005AA113120, the National Natural Science Foundation of China under Grant No.60473061 and 60473062.

We would like to thank to the anonymous referees for the useful comments on this paper and the suggestions given by them.

8. References

[1] Carlos E. Cuesta, María del Pilar Romay, Pablo de la Fuente, Manuel B: Temporal Superimposition of Aspects for Dynamic Software Architecture. *FMOODS 2006,* LNCS, pp. 93-107, 2006.

[2] Early Aspects homepage. http://www.early-aspects.net/.

[3] Aspect Oriented Software Development homepage. http://aosd.net.

[4] F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):1–38, 2004.

[5] R. J. Allen. A Formal Approach to Software Architecture. Ph.D. Thesis, Carnegie Mellon University, 1997.

[6] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, New Jersey, 1985.

[7] Formal Systems. FDR2 Homepage. http://www.fsel.com/software.html..

[8] Szyperski, C. Component software: beyond object-oriented programming. ACM Press and Addison Wesley, New York, USA (1998).

[9] Pérez. J., Ramos. I., Jaén. J., Letelier. P., Navarro. E. PRISMA: Towards Quality, Aspect-Oriented and Dynamic SoftwareArchitectures. In *Proc. of 3rd IEEE Intl Conf. on Quality Software (QSIC 2003)*, Dallas, Texas, USA, November (2003).

[10] Aldrich, J. Open modules: Modular reasoning about advice. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP'05)*, July 2005.

[11] Garlan. D., Monroe, R., Wile, D. ACME: An Architecture Description Interchange Language, *Proc. CASCON '97*, Nov. 1997.