# THE VIRTUAL ADAPTER – AN ABSTRACTION MECHANISM FOR WEAVING COMPUTING ASPECTS

*Chao-Hsin Lin*
Department of Risk Management and Insuracne,
National Kaohsiung First University of Science and Technology, Taiwan, R.O.C.
Email:linchao@ccms.nkfu.edu.tw

**ABSTRACT**

Although the aspect-oriented technology allows aspect weaver to weave aspect components across the function boundary, it seems that current aspect-oriented technologies are not handling the hardwired linkage well. The hardwired linkage can be typically found in reflection-based AOP approaches to date, and it may raise different issues regarding which kind of weavers (general-purposed or domain-specific) is used. This paper considers that the hardwired linkage is not a trivial issue, and might restrict the aspect-oriented concepts to be further practically applied to the large-scaled software development. To deal with the hardwired linkage issues this paper presents the adapter object model.

Keywords: AOP, Adapter, reflection computing, MOP

## 1. INTRODUCTION

By means of the link point the aspect-oriented technology allows aspect weaver to weave aspect component across the function boundary [9] and thus to reuse a group of functionality. Currently, there are generally two kinds of aspect weavers [12] – the domain-specific and the general-purpose aspect weaver. For the domain-specific weavers, its implementation cannot be further modified, but its weaving behavior can be indirectly instructed by several predefined keywords. In such an approach, it assumes that by careful analysis of the intended computing aspect, a well-predefined weaver can be enough to compose the functional component and the intended aspect component [10]. However, due to the closure of the domain-specific weaver to adopt new weaving functions, programmers are therefore again forced to interleave aspect code in relation to other domains into functional components. On the contrary the general purposed aspect weaver can be considered as open and allows programmers to manually customize their own weaving methods. It thus gives programmers better flexibility to compose different computing aspects. However, we have observed that the weaving implementation that customized by programmers are actually object-based (or object-specific), while the implementation of the domain-specific weaver can be considered as language-based since the composing implementation is carefully hidden from programmers and is able to be applied to set of objects. With the object-based weavers, programmers are thus restricted in applying AOP to the large-scale system developing, since programmers need to rewrite the same composing algorithm or implementation for each concerned object.

Based on the previous observation, the paper will examine the adapter approach in the context of AOP terminology to demonstrate its potential to be yet another approach to support the concept of AOP and the way of composing different concerned aspects.

The adapter approach was first experimentally implemented in the language Adapter++ [8] and was mainly proposed for object-oriented languages such as C++, Smalltalk and Java in which objects are not capable of expressing certain computing aspects (especially the dynamical intra-object scheduling [3]) in the specification time, such as defining the synchronization condition in the object interface. It distinguishes from other aspect-composing technology by that programmers can abstract and encapsulate the meta-level manipulation of customizing the aspect implementation into a new language construct, called virtual adapter. Conceptually, the adapter programming paradigm is supported by such architecture that consisting of two parts: 1) the meta level implementation, which allows programmers to customize the weaving methods, and 2) the virtual adapter, which abstracts the meta-level customization as a new aspect “type”, and from which programmers can derive child aspect component for composing different aspects.

In this adapter architecture, we are not concerned of the detail of how to syntactically design the virtual adapter or how to open the meta-level implementation – it can be provided by completely opening the whole language or just opening part of the language such as the interaction mechanism among objects, as current Adapter++ do.

What we concern is the relationship between virtual adapter and its supporting mechanism in the meta-level - in which the meta-level weaving implementation that customized by programmers can be modularized into a virtual adapter, and then be served for various functional objects to compose multiple aspects. In the following sections, the adapter approach will be discussed in greater detail and several Adapter++ examples will be presented to aid the discussion, which are organized as follow: In section 2, we will further investigate the hardwired linkage issue, which we found non-trivial when applying the AOP towards the

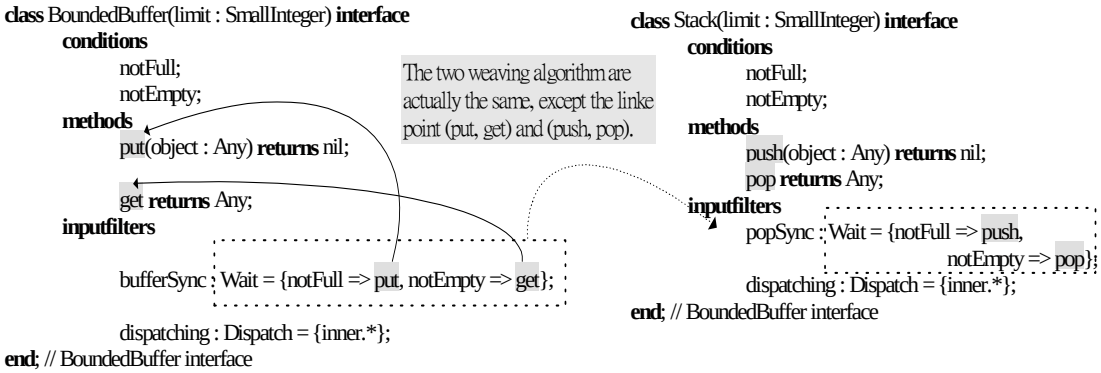


Figure 1 A bounded buffer example in Sina/ST

practice of software development. Section 3 will present the extend object model purposed in the adapter architecture. And then several Adapter++ examples are used to illustrate the weaving methodology in the purposed architecture. Finally, section 5 concludes this paper.

## 2. PROBLEM STATEMENTS

The reflective language also allows programmers to "indirectly" compose different aspects – programmers can first customize the language with new features via manipulating the language syntax and semantics, and then use it to compose new aspects and existing objects. However, one of the reasons that the AOP concept emerges is that it promises programmers a simplified method to compose different aspects. Therefore, instead of directly employing a reflective language for aspect composition, current approaches that supporting AOP are designated in a way, by which without the necessary of firstly modifying the syntax or semantics in the language level, programmers are able to treat the actions of composing different aspects as first class in the programming level.

With most of the AOP technologies to date, this paper recognizes the "name reference" as one of the import characteristics for aspect weavers to facilitate the simplification of the aspect composition. The name reference allows programmers to specify the link point by directly using names of different aspect components (such as the modular names for various functionalities or other concerns) for composition in the *weaving component* - in the paper, the weaving component is the place where programmers instruct or direct the weaver how to compose different aspect component.

Although direct name-reference have greatly facilitated the simplification of the aspect composition, it might cause the "hardwired linkage", which is the one of the main issues that the adapter approach is trying to deal with, and is discussed as follows.

Hardwired linkage. The hardwired linkage occurs when programmers use the name reference to specify the link point in the weaving component. The hardwired linkage can be best illustrated in Figure 1 using the language Sina/ST as an example. The language Sina/St provides programmers with a dedicated interface construct in which the conditions and methods that located in two different sections can be linked in the composing filters. For example, in Figure 1, the link point such as "notFull=>put" and "notEmpty=>get" is hardwired in the implementation body of the *bufferSyn* and *popSyn* respectively.

The hardwired linkage can be typically found in reflection-based AOP approaches to date, and it may raise different issues regarding which kind of weavers (general-purposed or domain-specific) is used:

- With general purposed weavers, the weaving components consist of two parts: the link point specification and the weaving mechanisms. For example, in Figure 1, the weaving components (e.g. composing filters) implement the necessary synchronization mechanism, where the hardwired linkage is also specified. As a result, although the objects BoundedBuffer and Stack are using the same algorithm for synchronization, further modularization and reuse of such weaving components for objects with each other is prohibited.

- On the contrary, with the domain-specific weaver, its mechanism for supporting aspect weaving of the intended computing domain is hidden from programmers and is not enclosed in the weaving components. For a domain-specific language example in Figure 2, the language D [10] allows programmers to associate the functional component (e.g., buffer) with the weaving component (e.g., bufferCoord), and the mechanism for the coordination of threads is not implemented by programmers but taken cared by the hidden aspect weaver. In this way, the domain-specific weaver effectively modularizes the underlying mechanism of the intended aspect that it can be reused for various functional objects. Nevertheless, with such approaches the hardwired linkage leads to another issues: It is difficult to provide programmers with proper methods to further customize their own domain-specific weavers, since it might involve the language-level manipulation of the syntax or semantics within the reflective language.

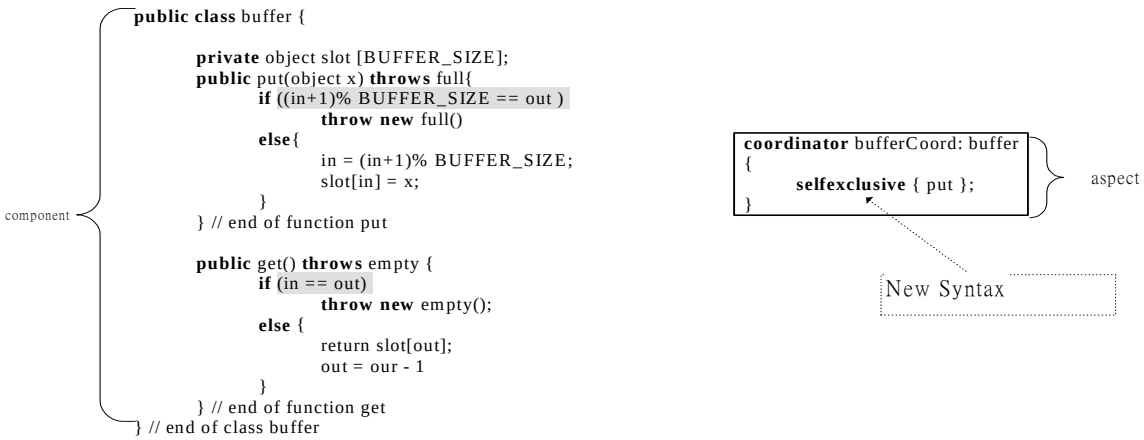


Figure 2 A bounded buffer example in language D.

Based on the previous discussion, we identify that it is the hardwired linkage phenomena that restricts the weaving component to be modularized and to be reused for various functional objects within the general-purposed weavers. It also leads to the difficulty of purposing proper as well as simplified methodology for programmers to design their own domain-specific weavers without involving into the language-level manipulation, syntactically or semantically.

We thus argue that by abstracting the hardwired linkage away from the weaving component it might be possible for providing programmers with yet another methodology to support the AOP concept.

In this paper, we argue that in the lack of proper abstraction mechanism the aspect-composing programming supported by current general-purposed aspect weavers is difficult to be modularized for language-wise reuse. For example, referring to

We also argue that, in the lack of proper abstraction mechanism, providing programmers with the capability of designing their own domain-specific weavers is also difficult to be relieved from directly manipulating the syntax in the language level, which is inconsistence with the simplicity issues that the domain-specific weavers try to deal with.

## 3. THE ADAPTER OBJECT MODEL

This extended object model, called adapter object model in this paper, consists of three parts: the *object* in base level, the *adapter* in the adaptation level, and the meta object in the meta level. The base-level *object* is mainly concerned with the functional behaviors related to applications such as the class in C++. The other two parts, adapter and meta object, are coupled to provide the necessary mechanism in the intended domains - when coupling the adapter and meta object, the adapter is used to encapsulate and abstract the meta-level manipulation in the meta object and it thus can be seen as an extended interface to the base-level interface construct.

Figure 3 shows the systematic way to exploit such an extended object model where the manipulation of the meta-level implemented by programmers (the shadow area) is abstracted to a virtual adapter (①) and is also dedicated to this adapter for implementing the necessary mechanism in the intended computing domain. The virtual adapter can be derived in order to associate with different functional objects (②)– with the purpose of introducing new aspects into this functional object.

Currently, the adapter object model is implemented in the language Adapter++, which is responsible to weave the

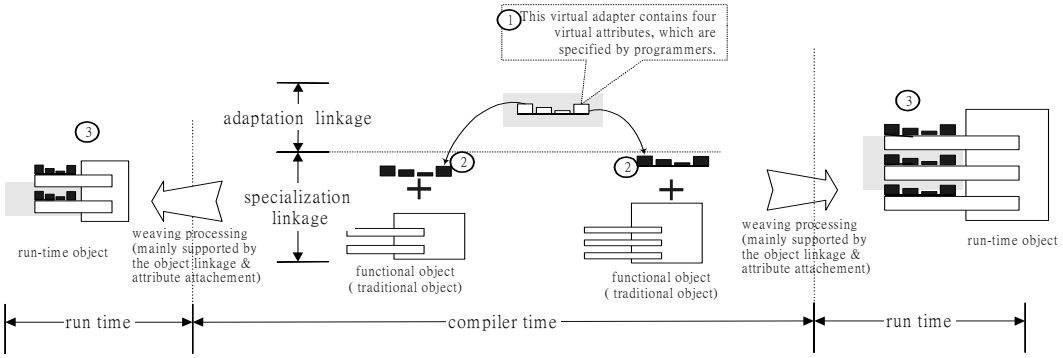


Figure 3 the extended object model

above three parts of specification into C++ objects (③). Two main weaving mechanisms are implemented to support weaving in such a model, the object linkage and the attribute attachment.

Object Linkage. The object linkage is established when the same name for base-level *class*, adaptation-level *adapter*, and meta-level *mclass* is declared, which is used to direct the Adapter++ where to find these three associated parts. In the adapter model, the object linkage can be specified at two different situations: one is for linking virtual adapter with its meta object (the shadow area in Figure 3); the other is for linking the functional objects with some concrete adapter, which is derived from some virtual adapter. In this paper, such object linkages are called as "adaptation linkage" and "specialization linkage" respectively:

- The adaptation linkage is for adding new features into object model and to forming the virtual adapter - the term "adaptation" is used to reveal that each virtual adapter represents a new extended object model.
- The specialization linkage is for traditional objects to adopting new features that encapsulated in this virtual adapter – the term "specialization" is used to symbolize that end users are able to specialize a generalized virtual adapter by concrete the regarding virtual attribute with respect to various functional objects.

Attribute Attachment. The attribute attachment mechanism is for programmers to choose and impose the virtual attribute that declared in the virtual adapter onto the functional behaviors while latter the specialization linkage is specified. Once the linkage is explicitly established, Adapter++ will weave these three associated parts across the separated level to generate a real object for run-time purpose, and such a weaving process is also aided by traveling through the imposing relationship between the attribute and the functional behavior that specified by programmers.

In the next section, an Adapter++ example is used to aid the illustration of how to exploit such virtual adapter for providing C++ objects with the synchronization capability. More detailed discussion about meta-level customization for such a synchronized adapter will be followed after presenting the default specification of the meta object.

## 4. EXAMPLE: THE VIRTUAL ADAPTER - SYNCHRONIZER

Although we choose the object-oriented language C++ for programmers to specify objects for the functionality in the application domain, this adapter approach is focused to be applied to other OO languages as well. Figure 4 shows the definition of the bounded buffer object (BoundedBuffer), which comprises two functions - *put()* and *get()* - for placing items into a limited buffer and retrieving items from the buffer respectively.

```
class BoundedBuffer {
        private:
        int    in, out, max, buf(SIZE);
    public:
        void    put(int x);
        int get( );
        BoundedBuffer( );
}
```

Figure 4 Bounded buffer object specification in base label.

In Adapter++, the new language construct *adapter* consists of three major components: namely the *imposing* section, *non-imposing* section, and *binding* section. In the imposing section, programmers declare virtual attribute functions for which its linked meta object that explicitly specified in the adaptation linkage is responsible to interpret. In the non-imposing section, those functions that are not interested to be imposed to the functional behaviors are specified here. The above two sections are both designated to be interpreted by its linked meta object in the adaptation linkage. On the contrary, the binding section is to be used in the specialization linkage. This binding section is for programmers to explicitly attach the virtual attribute function (declared in the imposing section in a virtual adapter) to the individual functional behavior of the associated host functional objects. In the following, Figure 5 and Figure 6 will be used to illustrate the regarding usage of the adapter in specialization linkage and adaptation linkage.

### 4.1. Specialization Linkage – Composing the Functional Behavior and the Aspect Attribute

The virtual adapter *synchronizer* (referring to Figure 6) has only one attribute that programmers can choose in the specialization linkage to impose onto the functional behaviors that declared in the associated host functional object. As shown in Figure 5, the adapter *BoundedBuffer* is derived from the virtual adapter *synchronizer* and given the same name as the class *BoundedBuffer* for the purpose of specialization linkage. To adopt the capability of synchronization into the functional object *BoundedBuffe*, the attribute *guardian()* is imposed onto each functional behavior - *put()* and *get()* - and is to be concreted by specifying the guardian attribute in the binding section for the non-full and non-empty condition respectively.

```
adaptation{    //switched to adaptation level
        adapter    BoundedBuffer : synchronizer{ // object linkage for
class BoundedBuffer & adapter BoundedBuffer
                binding:
                put(x) <- guardian( );
                get ( ) <- guardian ( );
        } // end of adapter buffer
        BoundedBuffer :: put(x) <- guardian( )        {// guardian
attribute for put to detect the not "full" condition
                return (bthis->in+1)%max != bthis->out;
        }
        BoundedBuffer :: get( ) <- guardian( )        { //guardian
attribute for put to detect the not "empty" condition
                return (bthis->in != bthis->out) ;
        }
```

```
adaptation }
```

Figure 5 The specialization linkage and attribute attachment.

### 4.2. Adaptation Linkage – Adding the Synchronization Feature into Object Model

To introduce the synchronization capability into the object model a new virtual adapter *synchronizer* is first defined, in which a virtual attribute *guardian* is declared in the *imposing* section, and an associated meta object named *synchronizer* is also derived from the default mclass *default* to support this virtual adapter *synchronizer*, as shown in Figure 6. In the following we will first present the mclass *default*, and then discuss the mclass *synchronizer*.

```
adaptation{   //switched to adaptation level
      adapter   synchronizer : default { //virtual adapter
            imposing:
                  virtual guardian();
            non-imposing:
                  // empty
            binding:
                  // Empty until specialization phase.
            }
adaptation}

meta{   //   switching to meta level
      mclass synchronizer :default {
            int getNextReadyOpId();
      }

      OperationId synchronizer::getNextReadyOpId() {
            OperationId behaviorId= Null;
            move2FirstEvent();
            behaviorId = qetCurrentBehId()
            for( ; behaviorId; ) {
                  if (athis->gurdian(behaviorId) )
                        return behaviorId ;
                  else
                        behaviorId = getNextBehID();
            } // end for
            return behaviorId ;
      }

meta}   //level switching
```

Figure 6 Adaptation linkage for adding synchronization into object model.

In Adapter++, the mclass *default* (Figure 7) implements the necessary Meta Object Protocol (MOP) for programmers to customize the adapter object model. The MOP is typically employed to specify a set of functions that represents the underlying meta-level implementation, by which users can customize the meta-level implementation. Adapter++, however, differs from others by that such a customizing process is encapsulated via the virtual adapter. In this way, end users can consider each virtual adapter as a different domain-specific weaver. Currently, the MOP functions in mclass *default* are mainly relative to the manipulation of the requesting events that are asking for services from an object, but are possibly suspended in the waiting queue. For example, functions *move2FirstEvent()*, *move2LastEvent()*, *move2NextEvent()*, *move2PreviousEvent()*, and *deleteCurEvent()* are all used to manipulate the movement of the event pointer in the waiting queue, and function *getCurrentBehID()* is used to retrieve the behavior ID for that current examined event is asking for.

The function *getNextReadyBehID()* plays an important role in the meta object part - it is used to tell the Adapter++ how to determine which behavior of the host object will be executed next by returning the behavior ID. In the mclass *default*, the *getNextReadyBehID()* is implemented to make the object process events in first in first out – as in most traditional object models.

```
meta {      // the default system specification of the meta object
  typedef OperationId int;
  mclass default {
      public:
      deleteCurEvent( );
      move2PreviousEvent( );
      move2NextEvent( );
      move2FirstEvent( );
      move2LastEvent( );
      BehaviorID getCurrentBehID ( );
      BehaviorID getNextBehID ( );
      BehaviorID getNextReadyBehID ( ) { //default is FIFO
            BehaviorID tempID;
            if (tempID = getNextBehID( ))
                  return tempID;
            else
                  return 0;
            };
      }
meta }
```

Figure 7 Specification of the Default meta object – mclass *default*.

To customize the default meta object for supporting the synchronization mechanism for virtual adapter *synchronizer*, a new mclass *synchronizer* (in Figure 6) is declared and implemented - which is accomplished by simply overriding the function *getNextReadyBehId()*. In the function *getNextReadyBehId()*, the event pointer is first moved to the first event in the waiting queue, and then functions *getCurrentBehID()* and *getNextReadyBehID()* will be used to retrieve the regarding behavior ID that current examined event is asking for. This behavior ID will be passed into *guardian()* as parameter that Adapter++ can dispatch the attribute call to the regarding *guardian* attribute - if the attribute function *guardian()* returns a value of 0 (false), the request will be left in the waiting queue and postponed; otherwise its ID is returned for execution by function *getNextReadyBehID()* and is also removed from the waiting queue. For example, when latter linked to object BoundedBuffer, the real guardian attribute could be the one that attached to function *put()* or *get()*.

Note that the guardian attribute in the adapter synchronizer (Figure 6) is referred by a new keyword “athis”. In C++, the keyword “this” is used to point to the object itself. In Adapter++, the keyword “this” is used to point to the object, adapter, or the meta object respectively, which is dependent on which level “this” is used. In addition, Adapter++ has two other keywords, “athis” and “bthis”. Keyword “bthis”, “athis” are used to refer to functional object part and adapter part respectively. With such key-

words, programmers can design how the three linked objects are causally connected.

### 4.3. Reuse of the meta-level mechanism

As mentioned previously, the virtual adapter is aimed to be used for modularizing and encapsulating the customization manipulation in the meta level, and thus to support the reuse of the customized meta-level mechanism. The main advantage of such design is that it allows programmers to avoid the hardwired linkage mentioned in section 2, and still simplifies the aspect composition. For the example in Figure 8, once the virtual adapter synchronizer is been provided, programmers can use it for object *Stack* to adopt the synchronization feature, in which end users will not be distracted to the meta-level implementation. What end users might concern is to 1) firstly specify the specialization linkage and then 2) attach the necessary synchronization condition for behavior *push()* and *pop()* respectively.

```
class Stack {
        private:
                int    in, out, max, buf(SIZE);
        public:
                void    push(int x);     // place an item into the buffer
                int gop( ); // remove an item from the buffer
                Stack( );                //    class constructor
}

adaptation{    //switched to adaptation level
        adapter    Stack : synchronizer{
                binding:
                push(x) <- guardian( );
                pop ( ) <- guardian ( );
        } // end of adapter buffer
        Stack :: push(x) <- guardian( )          {
                /* guardian attribute for put to detect the not "full"
                condition    checking the full condition: if buffer full,
                then return 0; otherwise return 1; */
        }
        Stack :: gop( ) <- guardian( )           {
                /*guardian attribute for put to detect the not "empty"
                condition checking the empty condition: if buffer
                empty, then return 0; otherwise return 1;*/
        }
adaptation }    //Switching back to base level.
```

Figure 8 Associating the virtual synchronizer adapter to class Stack.

## 5. CONCLUSION AND OPEN ISSUES

Although the aspect-oriented concept points out another possibility of code reuse by decomposing the software beyond the object functionality, it seems that, regarding to the discussion in this paper, current AOP technologies are not handling the hardwired linkage well. The hardwired linkage raises two types of problem: for general purposed weaver, the weaving component can not be modularized for the purpose of reuse; for aspect specific weaver, designating a reflective framework that allowing to be customized for new specific weavers is difficult without first opening a complex meta-level implementation in the language level – in other words, programmers can not directly focus on the aspect composition at first class, but need to first deal with the language-level syntax and even semantics for customize new weavers.

This paper considers that the hardwired linkage is not a trivial issue, and might restrict the aspect-oriented concepts to be further practically applied to the field of software engineering. To deal with the hardwired linkage issues this paper present the adapter object model, which suggests separating the object model into three parts, namely base object as well as adapter and meta object. The base object is used to define the functional behaviors as found in the traditional objects - since originally the adapter object model is designated to be applied in a number of traditional object-oriented languages. The meta object is to be used for defining the meta object protocol for communicating with the language implementation. In addition, the adapter is to be used as a new intermediate construct for encapsulating the meta-level mechanisms that customized in the meta object, and to be used for plugging into any base object for adopting new concerned aspects.

We have experimentally implement the adapter object model in the language Adapter++. In this paper, only one example of adopting the synchronization control into the functional object is presented. With this example, we believe that other instances of using such an adapter model can be inspired: For example, to equip an traditional (or functional) object with more complicated decision-policy capabilities of scheduling the arriving events, programmers can firstly implement the adapter-meta couple at the adaptation-linkage stage, in which the virtual attribute "priority" is declared in the adapter object and to be recognized and interpreted by its coupled (or linked) meta object for decision making; and then, at the specialization-linkage stage, this virtual adapter can be viewed as an specific-domain weaver in that programmers can attach each individual behaviors in the functional object to return different value, and thus support the aspect composition.

In conclusion, this paper presents the adapter object model for not only dealing with the hardwired linkage, but also having a higher-level goal - to support the application of aspect oriented concept towards the large-scaled software development. In this model, what we concern is not how to build the meta-level implementation, or what kind of object-oriented languages should be used as the base-level language - Designers can choose to build their own meta-object protocols in the meta level, and choose different target object-oriented languages for the base-level language. Rather, what we really concern is that how to provide programmers with the capability of abstracting and encapsulating the meta-level customization that programmers can specify the object in two dimensions: one for functional behavior, and the other for special computing aspects such as synchronization conditions for concurrent computing.

## 6. REFERENCES

[1] Aksit, M. and Tripathi, A., "Data abstraction and mechanism in Sina/ST", Proceeding of OOPSLA'88,

volume 23, pages 207-275, SIGPLAN Notices, ACM Press, 1988.

[2] Chiba, S., “A Metaobject Protocol for C++”, Proceedings of OOPSLA’95, SIGPLAN Notices, Vol. 30, No. 10, Austin, TX, ACM, pp. 285-299, 1995.

[3] Elrad, T., “Comprehensive Race Controls: A Versatile Scheduling Mechanism for Real-Time Applications,” Proceedings of the Ada Europe Conference, n.pag., Madrid, Spain, June 1989.

[4] Frolund, S., “Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages,” ECOOP ‘92 Proceedings, pp.185-196, June 1992.

[5] Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezka, H., and Kubota, K., “Design and Implementation of Metalevel Architecture in C++ - MPC++ Approach,” Reflection Symposium’96, San Francisco, CA, n.pag, April 21-24, 1996.

[6] Kafura, D.G., Mukherji, M., and Lavender, G., “ACT++ 2.0: A Class Library for Concurrent Programming in C++ Using Actors,” JOOP, Vol. 6, No. 6, pp. 47-55, 1993.

[7] Kiczales, G., “Beyond the Black Box: Open Implementation”, IEEE Software, pp. 137-142, 1996.

[8] C.H. Lin, E. Tzilla, “An Enhanced Reflective Architecture for Adaptation of Object-Oriented Language/Software”, The proceeding of Asia-Pasific Software Engineering Conference, IEEE, pp. 20- 27, 1998.

[9] Lopes, C. V. and Hürsch, W. L., “Separation of Concerns”, Tech Report of College of Computer Science, Northeastern University, Boston, MA 02115, USA, Feb 24, 1995.

[10] Lopes, C.V., Kiczales G., “D: A Language Framework for Distributed Programming”, Xerox PARC, Palo Alto, CA. Technical report SPL97-010 P9710047, February 1997.

[11] Matsuoka, S., and Yonezawa, A., “Analysis of Inheritance Anomaly in Object-Oriented Languages,” Research Directions in Object-Based Concurrency ed. G. Agha, P. Wegner, A. Yonezawa, The MIT Press, Cambridge, MA, pp.107-150, 1993.

[12] Mens, K., Lopes, C.V., Tekinerdogan B., Kiczales G., “Aspect-Oriented Programming WorkShop Report”, Proceeding of the Aspect-Oriented Programming WorkShop at ECOOP’97, June, 1997.

[13] Papathomas, M., and Nierstrasz, O.N., “Supporting Software Reuse in Concurrent Object-Oriented Programming Language: Exploring the Language Design Space,” Object Composition, ed., D. Tsichrizis, pp.189-204, University of Geneva, 1991.

[14] Snyder, A., “Inheritance and the Development of Encapsulated Software Systems,” Research Directions in Object-Based Concurrency, eds., G. Agha, P. Wegner, A. Yonezawa, The MIT Press, Cambridge, MA, pp. 165-188, 1993.

[15] Tomlinson, C., and Singh, V., “Inheritance and synchronization with Enabled-Sets”, Proceeding of OOPSLA ‘89, volume 24, pp. 103-112, SIGPLAN Notices, ACM Press, 1989.

[16] Yonezawa, A., Shibayama, E., Takada, T., and Honda, Y., “Modeling and Programming in an Object-Oriented Language ABCL/1,” Object-Oriented Concurrent Programming ed. A. Yonezawa and M. Tokoro, The MIT Press, Cambridge, MA, pp. 55-90, 1993.