1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Adaptation of the Object-Oriented Language/Software in Adapter++

Chao-Hsin Lin

Department of Risk Management and Insurance
National Kaohsiung First University of Science and Technology, Kaohsiung, Taiwan, R.O.C.
E-mail: linchao@ccms.nkfu.edu.tw

ABSTRACT

Recently, instead of building compilers from scratch, "open language" further allow programmers to customize their own new interface constructs to support special requirements and object-oriented requirements at the same time. This paper presents our language adapter++, which is implemented based on the concept of open implementation that the object model can be extended to address different requirements in various computing environments. Adapter++ is distinct from other open languages by that the implementation of extended object model is not directly reified into the syntax or semantics of interface construct to become a new integrated construct but into a separated interface. Programmers can then use the two interfaces to define an object as if using integrated constructs and thus effectively avoid the inheritance exclusion and inheritance anomaly.

1. INTRODUCTION

In sequential object-oriented languages, the interface construct (e.g., class in C++) providing information hiding and data abstraction that programmers can view the construction of system as a collection of interacting software objects that reflect real world counterparts. Traditionally, the interface construct of object also effectively relieve programmers the task of reusing existing code, since this mechanism means that programmer's attention is no longer distracted by irrelevant implementation details. However, based on the need of dealing with the special concerns in various computing domains, a number of researchers try to add new notations into traditional sequential interface constructs to better support object-oriented programming in different computing environments [2, 4, 6, 8, 10, 13, 18, 21]. For instance, to support concurrent programming, concurrent C/C++ [18], ABCL/1 [31], Ada [15] and Eiffel // [9] introduce unique linguistic support to manage communication and interaction among objects. In [24], such a new object interface-construct is called integrated construct, which allows programmers to define functional behaviors and aspect-related behaviors in the same interface construct for an object. Within such an integrated interface, the aspect-related behaviors such as concurrency controls for concurrent programming, timing constraints for real-time programming, and location control for distributed programming are usually distinguished from the functional behaviors for application domains by special new language notation. Nevertheless, using integrated construct usually leads to inheritance anomalies (interference of inheritance between aspect-related behaviors and functional behaviors [1, 2, 31]) and inheritance exclusion (different interface constructs can not inherit from each other [24]). In addition, integrated constructs also lead to black box problems [23], since programmers are no longer allowed to modify the mechanism embedded in language semantics or syntax.

This paper presents our adaptable architecture, which is implemented in language adapter++. The adaptable architecture is designed in a way that programmers can adapt the existing single object or a group of related objects to different computing environments without changing or rewriting. Furthermore, the adaptation architecture can be modified to improve the language itself; and the resulting language better takes care the inheritance anomaly and the inheritance exclusion than other approaches of open languages.

Section 2 introduces more concepts underlying our adaptation architecture in details and the other related works. Section 3 presents an overview of our proposed adaptation architecture for object-oriented language and the design concepts in the language adapter++. Section 4 introduces the programming methodology of using adapter++ to support language adaptation via the adaptation phase and software adaptation via a specialization phase. The bounded buffer example will be used to further illustrate how to employ the proposed two-phase programming methodology in concurrent programming. It provides the general idea of adapting software/language to other computing environments with adapter++. Finally, section 5 concludes our research.

2. BACKGROUND AND RELATED RESEARCH

Without integrated construct, programmers need to design their own low-level software primitives to support development of software in different computing environment. For instance, while using language Eiffel in concurrent programming, programmers need to employee the underlying Eiffel assertion mechanism to ensure mutual exclusive access to shared data [9, 22, 26, 32]. In this way, synchronization constraints (e.g., pre-conditions) are closely tied to method codes and thus possibly leading to one of the inheritance anomaly - the complete rewriting of the previously defined synchronization controls and method codes when introducing new methods into the sub-classes. To separate the code for special concerns from the method body, Concurrent C/C++ [18], ABCL/1 [31], Ada

[15], RTC++[20] and Eiffel // [9] all provide programmers with integrated constructs, in which special purpose code can be directly specified in integrated construct to better support code reuse via inheritance.

In recent years, opening language further allowing programmers to customize the integrated construct to simplify the object-model reasoning and thus facilitating the application development in various computing environments has received considerable attention [2, 11, 19, 23, 27, 31, 41]. The idea of Open Implementation is to expose an abstraction of the implementation, not the actual details. Kiczales [23] claims that blackbox abstractions by using the object model do not always work, because it is impossible to hide all implementation issues behind a module interface. The Open Implementation must therefore provide clients with control over its implementation strategy, not with a mountain of details [23]. Under these approaches, users usually consider the object-oriented language in two different levels – base level and meta level. Conceptually, programmers can modify the meta-level objects to affect how the base-level objects interact with each other such as synchronization control, timing constraint, assertion code of debugging, and location directives of network objects. Conceptually, programmers can extend the language by means of modifying the meta-level objects. The remainder of this section will discuss several open languages and also present the criteria of designing an open language.

Open C++ is extensible in language semantics. Programmers can add new language primitives and thus changing the language semantics to support special requirements in different computing environments [11]. However, by lacking of the capability of allowing programmers to customize the integrated construct, several issues of software engineering such as software designs, software analysis and modular programming is restrictedly supported.

To support software design and module programming, for a declarative language such as C and C++, language semantics as well as syntax extensions are required to be open or be reflective. The MPC++ meta-level architecture was designed and implemented to meet the above requirements [19]. The MPC++ defines an abstract compiler, which consists of a lexical analyzer, parser, and code generator. By means of the abstract compiler, the semantics and syntax of an imperative language like C or C++ can be extended. However, MPC++ only allow programmers to design traditional integrated construct that the issue such as inheritance anomalies and inheritance exclusion are ignored.

The CodA [28] meta-level architecture is based on an operational decomposition of meta-level behavior into objects and the provision of a framework for managing the resultant components. For example, the interaction between objects can be decomposed into accept, queue, receive state, state, and execution.

Open C++, MPC++, CodA only focus on providing programmers with the capability of customizing languages but not on avoiding the inheritance anomaly and inheritance exclusion. Another open language ABCL/R2 [31] trying to avoid inheritance anomaly by providing its special integrated construct with multiple concurrency control schemes that programmers can choose the suitable scheme to avoid code rewriting. However, ABCL/R2 does not focus on allowing programmers to invent new integrated construct.

Although open implementation allows programmers to design suitable integrated construct to resolve the blackbox problems, it has been reported that developing applications by means of employing the integrated construct approach will still suffers the inheritance anomaly in several cases [31]. Another issue of code reuse of employing the integrated construct is inheritance exclusion [24], which is not found in pure object-oriented language. For examples, in language Concurrent C++, capsule (a kind of integrated construct used to protect the integrity of shared data) can not inherit from class, and vice versa. Because of inheritance exclusion, the object code is not transportable among different computing environments and even again lead to the completely rewriting of object code. We therefore consider that the criteria of an open language should also regarding that the extended language can support high-level programming and provide programmers with the capability of effectively avoiding inheritance anomaly and inheritance exclusion.

3. THE PURPOSED ADAPTABLE ARCHITECTURE

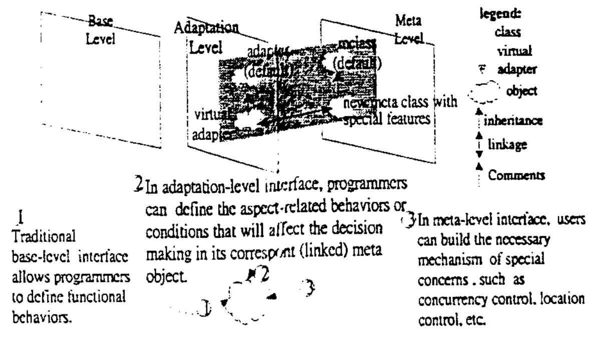
Base Level
Adaptation Level
adapter (default)
mclass (default)
Meta Level
virtual adapter
new meta class with special features
legend:
class
virtual
adapter
object
inheritance
linkage
Comments
1 Traditional base-level interface allows programmers to define functional behaviors.
2 In adaptation-level interface, programmers can define the aspect-related behaviors or conditions that will affect the decision making in its correspont (linked) meta object.
3 In meta-level interface, users can build the necessary mechanism of special concerns. such as concurrency control, location control, etc.
1
2
3

Figure 1 Overview of the enhanced reflective architecture in Adapter++

Our approach differs from most existing reflective approaches [2, 11, 19, 23, 27, 31, 41] specifically in the incorporation of an additional intermediate abstraction level (between the base level and meta level (Figure 1). It means that users can view an object in three interfaces: base-level interface, adaptation-level interface, and meta-level interface. In this way, programmers can extend the language and also avoid the inheritance anomaly and the inheritance exclusion while using the resultant language.

This arrangement promotes the principle of *separation of concerns* since each level deals with only a specific functionality of the object entity. The *base level* is mainly concerned with the functional behaviors related to applications. The adaptation level deals with aspect-related behaviors that irrelevant to functional behaviors but relevant to the underlying computing of an special purpose object model. The *meta level* implements the object model itself as presented in other reflective architecture.

In the following sections, we will introduce some key concepts that will be used throughout the paper.

3.1. Inheritance mechanism and linkage mechanism

In the proposed architectural design (Figure 1), two types of inter-class hierarchy relationships are presented. One is the inheritance relationship as in the traditional object-oriented languages. In this case, each of the three abstraction levels (*the base level*, *adaptation level*, and *meta level*) has its own independent inheritance hierarchy. These sets of hierarchies serve to support code sharing in defining new sub-classes.

Inheritance mechanism is used to inform a compiler where to find the parent classes for the inherited features. It deals with the vertical relationships among objects within their respective class hierarchies. However, the crossed inheritance relationship is not allowed since it's not necessary.

Another inter-class hierarchy relationship is the *linkage mechanism*. A linkage is established when the same class name is used for the *base-level* object and the corresponding *adapter* and *meta-object*. Such a design will not lead to name ambiguous, since each level has its own name space. The proposed *linkage mechanism* is mainly used to assist the compiler in gathering needed information found in the three associated parts (class, adapter and metaobject). Once the linkage is established, Adapter++ is responsible to weave linked objectst together into a real object in the run time.

3.2. The base level

We choose the object-oriented language C++ as the default *base-level* language because it has been recognized as a very popular general-purpose programming language. However, this adaptable architecture can be applied to other (object-oriented reflective) languages as well. In our design, a *base level* object definition usually consists of a data specification and method implementation as in traditional object-oriented languages.

3.3. The adaptation level

At the adaptation level, a new language construct (*adapter*) is provided as an association link between the base-level object and its respective metaobject(s). A *virtual adapter* hides the underlying (complicated) computation performed by the corresponding metaobject(s) from programmers. An *adapter* can be regarded as an extended interface to the traditional data abstraction construct (e.g., C++ class). The functionality implemented in an *adapter* permits the incorporation of special user requirements in the *base-level* object. These requirements may cover most of the special requirements in different computing environments such as synchronization conditions, real-time constraints, etc. This information is then managed by the *meta-object* component to monitor and regulate the behavior of the associated object entity.

```
adaptation {    // switch to adaptation level
      adapter   default {   // In this root-class all the
following sections are left unspecified
              non-imposing:
                     // Empty section.
              imposing :
                     // Empty section.
              binding:
                     // Empty section.
      } // adapter
   } adaptation   // switch   out to non-adapted base language
```

Figure 2 Specification of default adapter called default.

An *adapter* consists of three major components: namely the *imposing* section, *non-imposing* section, and *binding* section (See Figure 2). Figure 2 presents the default adapter, which contains no entry in the three sections. When the base-level objects are not explicitly associated with any *adapter*, the default adapter and its correspondent (default) meta object will be assumed:

(1) Imposing Section. Functions declared in this section are termed imposing functions. They serve to impose certain constraints (e.g., synchronization constraints) or notations on the functional behaviors found in the base-level object. For instance, the guardian constraints for synchronization can be declared here.

(2) Non-imposing Section. All other local operations that are not classified as imposing functions are specified in the *non-imposing* section.

(3) Binding Section. In this section, users can explicitly specify which imposing functions are to be associated with the selected functional behaviors of a base level object. This section is usually empty until a concrete adapter is linked to a given base-level object, users can then complete all the implementation details of these imposing functions (see section 0).

3.4. The meta level

In many reflective languages, meta-objects exist at run-time or compiler time to manage the association of base-level objects. The Meta-Object Protocol (MOP) is often employed to specify the implementation of a reflective object-model which users may extend or modify. Essentially, a MOP specifies the implementation of a reflective object-model. Thus, a MOP specifies which meta-level objects are needed to implement an object model, be it as a consequence of reification or a meta-level declaration. Our architecture as other researches allows programmers to modify the meta object and thus adapting the object model to special purposes.

```
meta { // the default system specification of the meta object
    typedef OperationId int;
    class synchronization :default
        public:
            qDelCurrent( );
            qMvPrev( );
            qMvNext( );
            qMvFirst( );
            qMvLast( );
            OperationId getNextOpId ( );
            OperationId getNextReadyOpId ( ) {
                    return getNextOpId( );
            };
            int timing( int timeLimit) { return -1};
            void preAction( );
            void postAction( );
    }
meta }
```

Figure 3 Specification of the Default Meta-Object.

Figure 3 presents the default specification for the *meta object*. Function getNextOpId() retrieves the first event in the waiting queue and return the identify number of the requested operation (behavior). In default, function getNextReadyOpId() calls getNextOpId() and then return the value back. Such a design makes the object process events in a way of first in first out. The function getNextReadOpId() is usually been overloaded to design new scheduling policy. Function qDelCurrent() delete the current event that are been examined from the waiting queue. Function timing(int) is used for real-time programming. By default, timing() return negative integer to disable the real-time option. Function preAction() and PostAction() are used for location control in distributed computing. Functions qMvFirst(), qMvLast(), qMvNext(), qMvPrev(), and qDelCurrent() are all used to manipulate the waiting queue.

In addition, the *meta object* could be an active object or a passive object. It depends on if a function engine() exists in derived meta object.

3.5. Implementation of Adapter++

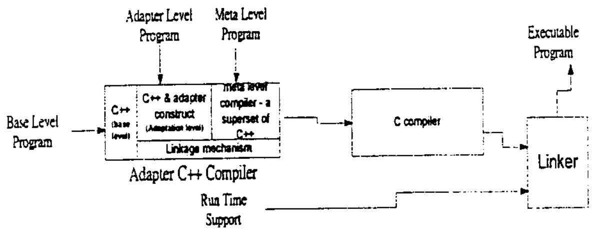

Figure 4 Adapter++ Compiler

The compiler that we built consists of C++ for base-level programming, a superset of C++ at the adaptation level and ConcurrentC/C++ at the meta level.

The source code is also divided into base level, adaptation level and meta level code. Source code at each level is compiled by the corresponding compiler.

3.5.1. *Syntax for Level Switching*

```
(1) base {            // Starting the base-level programming.
(2) base }            // Ending the base-level programming.
(3) adaptation }     // Starting the adaptaiton-level programming.
(4) Adaptation {     // Ending the adaptaiton-level programming.
(5) Meta {            // Starting the meta-level programming
(6) meta }            // Endign the meta-level programming
```

Figure 5. Syntax of level switch

Figure 5 shows how the syntax can be used to switch between different levels. To switch to the adaptation level, programmers will use the syntax represented in line three of Figure 5. To end the programming at the adaptation level, programmers will use the syntax specified in line four of Figure 5. In addition, the syntax specified at line 1 to 2 of Figure 5 can be used to switching in and off the base level while the syntax (line 5 to 6) is used to switch in and off the meta level.

3.5.2. *Name Resolution at All Three Levels.*

Since our object model is separated into three levels, objects with the same name will exist in different levels. To avoid the confusion, each level should have an independent name table of name resolution. When the statement of level switch is met, Adapter++ will select the correspondent table for name resolution.

3.5.3. *Implementation of the Linkage Mechanism.*

In compiler time, classes that declared in different levels but with the same name will be linked. The linkage mechanism in Adapter++ actually re-names these thre classes and further weaves them together into another class that will be instantiated in run time.

3.5.4. *Implementation for the Attaching Mechanism.*

```
ATTACH        :  <-
attach        :  ATTACH ID arg_list;
decl          :  decl   attach
              |  decl arg_list
              |  Del LP RP
              |  fname;
```

Figure 6. Syntax of attaching imposing function

Attaching mechanism, like "slot" in language Lisp, allows programmers to assign various attributes (or constraints) to any individual operation within a base-level object. The syntax is shown in Figure 6 and the example is presented in Figure 14.

3.5.5. *Run Time Overview*

Adapter ++ will weave the linked classes at the different levels (base-level class, adaptation-level class, and meta-level class with the same name) into C++ code. Then, the result code will be saved in an intermeddle file and can be further compiled by C++ compiler.

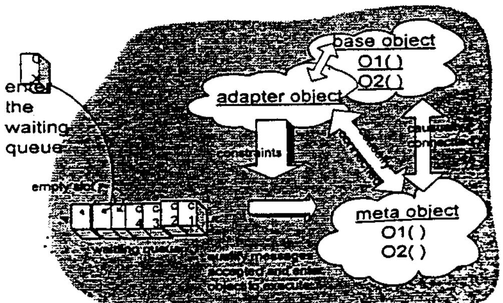

Figure 7. Run-time overview of linked objects

For each class defined at base-level, Adapter++ will look for its adaptation-level adapter and meta-level class to generate the woven code that can be used to generate the run-time metaobject. Conceptually, such a run-time run-time object consists of four sub-objects – the object that instantiated from base-level class, the object that instantiated from adaptation-level adapter and the object that instantiated from meta-level class, and the waiting queue object. In addition, these above objects are causally connected with each other, except the waiting queue (see Figure 7). It means that if the status of one of these three objects is changed, it leads to the change of the status of other two objects.

3.5.6. *Pointer to be used to point to object itself- "this", "bthis", "athis", and "mthis"*

In C++, the keyword "this" is used to point to itself. In Adapter++, the keyword "this" is also used to point to itself. In addition, Adapter++ has three other keywords, "athis", "bthis", and "mthis". Keyword "bthis", "athis", "mthis" are used to point to base-level object, adaptation-level adapter and meta-level object respectively.

With such keywords, programmers can design how the three linked objects are causally connected that the computational reflection can be present at run time.

Restriction rules are made for these pointer: 1) all three pointer can not be used in base level, 2) "mthis" can not be used in adaptation level, and 3) all three pointer can be used in meta level. With such a design, base-level programming is independent from the other two levels and the adaptation programming is independent from meta level architecture. In other word, the base-level program is saved from rewriting while the programming in other two levels are changed and the adapter in adaptation level will not change while the underlying meta-level architecture is modified.

4. LANGUAGE ADAPTATION AND SOFTWARE ADAPTATION WITH ADAPTER++

In this section, we present a programming methodology that uses the proposed architecture to support open implementation and language adaptation. The first phase is termed the *abstraction phase*; it is followed by the *specialization phase*.

4.1. The adaptation phase (phase I)- designing an extended interface

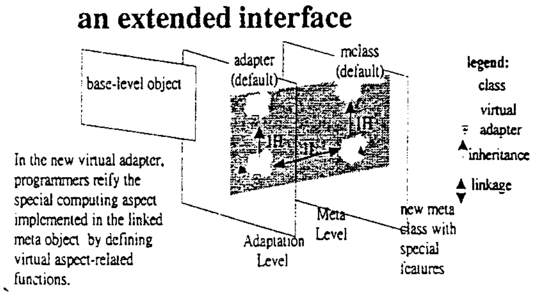
base-level object
adapter
(default)
mclass
(default)
1H
1L
1H
legend:
class
virtual
adapter
inheritance
linkage
In the new virtual adapter, programmers reify the special computing aspect implemented in the linked meta object by defining virtual aspect-related functions.
Adaptation
Level
Meta
Level
new meta class with special features

Figure 8 Pictorial representation of the abstraction phase.

At the *abstraction phase* (or phase I), users need to define an *abstract adapter* (*virtual adapter*). An *virtual adapter* can be seen as an extension interface to the *base-level* class. It differs from the base-level class in the types of functions (behaviors) being specified: the base-level class allows programmers to specify those functional behaviors for the application; those (aspect-related) behaviors irrelevant to object functional behaviors but for special concerns should be specified in virtual adapters.

One of the reasons that we separate the aspect-related behaviors from meta level is that such behaviors is usually dependent on the context of base-level classes. If not separated, the meta-level architecture may be forced to be re-written, once linked to another target base-level class.

Another reason is that the virtual adapter can hide the meta-level architecture from users as the integrated construct hiding the underlying language semantics from users. In this way, programmers can only consider the virtual adapter as an extension interface to the base-level class to define an object without caring its underlying meta-leve architecture in specialization phase (section 4.2).

In Figure 8, the shaded area presents the relationship of related objects in adaptation phase. The arrow, labeled as 1H (phase 1, inheritance), of the adaptation level in the shaded area illustrates that a new virtual adatpter for special purpose is derived from the system default adapter (named *default*). A corresponding specialized *meta class* is also derived (the arrow labeled as 1H of the *meta-level*) and is then linked to its target adapter as illustrated by the horizontal bi-directional arrow 1L (phase1, linkage) in Figure 8.

In the new virtual adapter, programmers reify the special computing aspect implemented in the linked meta object by defining virtual aspect-related functions. For example, programmers can provide a virtual guardian control function in a virtual adapter and implement the related mechanism in meta level object.

4.2. The specialization phase (phase II) - using the extended interface

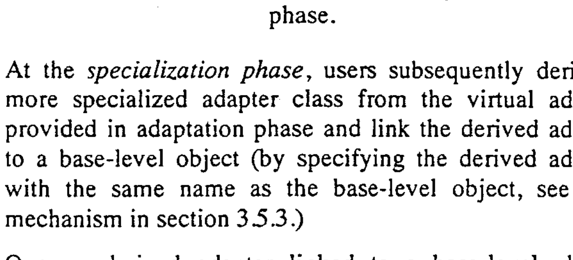
In specialization phase, programmers first derive child adapter from the virtual adapter provided in adaptation phase and then link it to base-level objects.
Once a derived adapter linked to a base-level object, programmers are then responsible to concrete the inherited virtual aspect-related behaviors in derived adapter.
adapter
(default)
mclass
(default)
virtual
adapter
2H
2L
linked
adapter
legend:
class
virtual
adapter
inheritance
linkage

Figure 9 Pictorial representation of the specialization phase.

At the *specialization phase*, users subsequently derive a more specialized adapter class from the virtual adapter provided in adaptation phase and link the derived adapter to a base-level object (by specifying the derived adapter with the same name as the base-level object, see link mechanism in section 3.5.3.)

Once a derived adapter linked to a base-level object, programmers are then responsible to concrete the inherited virtual aspect-related behaviors declared in adaptation phase.

The upward arrow labeled 2H in Figure 9 represents an inheritance relationship between the newly derived *adapter* and its ancestor, the virtual *adapter*. At this point, users can complete the implementation details of new adapter components by attaching the *imposing* functions to base-level behaviors in binding section. The horizontal bi-directional arrow labeled 2L in the shaded area of Figure 9 illustrates such a linkage association.

4.3. An Example of Using Adapter++ to Support Concurrent Programming

In this section, we provide a producer/consumer example to illustrate how to reuse a base level object in a concurrent environment. Figure 10 presents the pictorial representation of the inheritance and linkage relationships among the various object classes (e.g., *consumer*, *producer*, and *buffer* objects). In this figure, descriptions given in parentheses are the instance names of objects (e.g., *buffer*, *producer* and *consumer*). The descriptions above these names are their corresponding data types (e.g., *class*, *adapter* and *meta class*). These horizontal bi-directional arrows represent linkage relationship among three associated classes. The vertical arrows denote inheritance relationships between the parent classes and their corresponding descendant classes at each level of concerns. Numberings 1 and 2 represent the two phases of adaptation programming (namely the *Abstraction Phase* and *Specialization Phase* respectively).

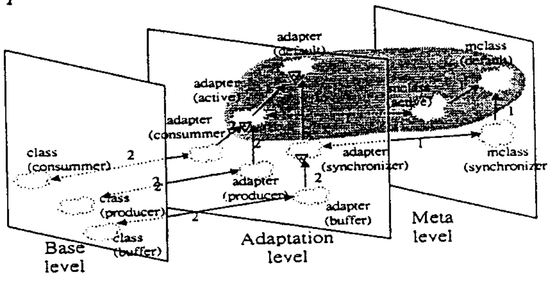

Figure 10 Inheritance and linkage relationships for the consumer, producer and buffer.

Figure 11 provides the *base-level* class definitions for the shared object *bounded buffer*. The new object is specified in the same manner as in C++.

```
Class bounded_buffer {
    Private:                // denoting shared resources
            int   in, out, max, buf(SIZE);
    public:
            void  put(int x);     // place an item into the buffer
            int get( );     // remove an item from the buffer
            buffer( );            //   class constructor
}
```

Figure 11 Bounded buffer object specification in base level.

4.3.1. *The adaptation phase - an interface for specifying synchronization conditions*

At the *adaptation Phase*, a new virtual adapter *synchronizer* (see Figure 12) is first formulated to provide synchronization control for the *base-level* object *bounded buffer*, which is not capable of supporting concurrent operations. The virtual function *guardian* (declared in the *imposing* section) acts as a constraint to be imposed on all application domain behaviors of the linked base-level object.

```
/*Abstraction Phase I: define new virtual adapter from which
more specialized   adapter could be derived. */
adaptation{ // Switching to adaptation level
    adapter  synchronizer : default
       imposing:
         virtual guardian( );// newly defined constraint function
       non-imposing:
    // empty
       binding:
    // Empty until specialization phase.
              }
  adaptation}   // Switching   back to base level
```

Figure 12 A new virtual *adapter* synchronizer.

After defining the new virtual adapter, a corresponding *meta level object* (Figure 13) must be defined. It is accomplished by treating the new *meta class synchronizer* as the direct descendant of the system *meta* class *default*. Note that both the newly derived *adapter* and *meta* class share the same class name.

In the new *meta class*, the inherited *meta-level* method *getNextReadyOpId()* is locally overridden to support the object-wise synchronization capability. This extended method will first examine elements (representing pending client requests) in the queue by activating the functions related to manipulate the waiting queue. In the example, if the imposing function *guardian* returns a value of 0 (or false), the request will be postponed; otherwise it is returned and removed from the queue for execution.

```
meta{   //level switching
    class synchronization :default
            int getNextReadyOpId();
    }
    OperationId synchronization::getNextReadyOpId() {
            qMvFirst();
            for( ; not the end of the queue; ) {
                    operationId = getNextOpId();
                    if ( athis->gurdian(operationId) ) {
                            qDelCurrent();
                            return operationId;
                    }
                    else
                            qMvNext();
            } // end for
            return 0;
    }
  meta}   //level switching
```

Figure 13 The correspondent meta-level of *Adapter* synchronization.

4.3.2. *The specialization phase*

```
/* Specialization Phase:   define a new adapter to be associated
with a selected base-level object and specify the binding
relationship between the inherited imposing functions and their
corresponding base-level member methods. */
adaptation{
  adapter   buffer : synchronizer{ // new adapter
         binding section:
         put(x) <- guardian( );
         get( ) <-   guardian ( );
  } // end of adapter buffer
  buffer :: put(x) <-   guardian( )       {
         return (in+1)%max != out;
  }
  buffer :: get( ) <-     guardian( )       {
         return (in !=out) ;
```

```
    }
adaptation }   Switching back to base level.
```

Figure 14 Establishing the binding relationship between the new adapter and its associated base-level object specification.

In the *Specialization Phase*, a user can define a more appropriate adapter to be associated with the *base-level* object *buffer* (see Figure 10). The new adapter *buffer* inherits the synchronization control capability previously stated. Here, we only need to specify the binding relationship between the inherited constraint patterns (e.g., the *imposing function* guardian controls) and their corresponding *base-level* member methods (e.g., methods *get* and *put*).

5. CONCLUSIONS

We have discussed two obstacles that lead to difficulty of reusing the existing software modules while applying the object-oriented technology to develop application in various computing domains: inheritance anomaly, it prevents users from building new objects by reuse existing objects; inheritance exclusion, it prevents users from adapting existing software or object to different computing domains.

To address these problems, we developed a three level architecture, a new technique that can be applied to various target object-oriented languages. Our proposed adaptable architecture is already applied to the language - Adapter++. The power of using Adapter++ in concurrent programming is already demonstrated in this paper. We also believe that the Adapter++ can be applied to other computing environments with the same flexibility.

This design of the three-level architecture offers the following advantages:

1. an unifying way to introduce different computing requirements into object model and thus to support the reuse of extended code,
2. the object code can be reused in different computing environments and thus support software adaptation by just changing the attaching adapter but without changing the object itself,
3. high-level programming is supported in the resultant extended languages.

6. REFERENCES

[1] Aksit M., and Bergmans L., “Obstacles in Object-Oriented Software Development”, Proc. of the OOPSLA'92 Conference, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 341-358, 1992.

[2] Aksit, M., Bosch, Vaudun, W., Sterren, J., and Bergmans, L., “Real-Time Specification Inheritance Anomalies and Real-Time Filters”, Proceeding of the ECOOP '94 Conference, LNCS 821, Springer Verlag, July 1994, pp. 386-407, 1994.

[3] America, P., and Linden, F., "A Parallel Object-Oriented Language with Inheritance and Subtyping," ECOOP/OOPSLA '90 Proceedings, pp. 161-168, 1990.

[4] Andrews, G.R., and Schneider, F.B., "Concepts and Notation for Concurrent Programming," ACM Computing Surveys, Vol. 15, No. 1, pp. 3-43, 1993.

[5] Bennett, J., "The Design and Implementation of Distributed Smalltalk," OOPSLA'87 Proceedings, pp. 318-330, 1987.

[6] Bos, J., and Laffra, C., "PROCOL: A Parallel Object-Oriented Language with Protocols," OOPSLA '89 Proceedings, SIGPLAN Notices, Vol. 24, No. 10, pp. 95-102, 1989.

[7] Briand, L., "Ada Real-Time Systems and Basic Priority Inheritance," ACM Ada Letters, Vol. 14, No. 3, pp. 105-112, 1994.

[8] Buhr, P. A., Dichfield, G., Stroobosscher, R.A., and Younger, B.M., "uC++: Concurrency in the Object-Oriented Language C++," Software-Practice and Experience, Vol. 22, No. 2, pp. 137-172, 1992.

[9] Caromel, D., “Toward a Method of Object-Oriented Concurrent Programming,” *Communications of the ACM*, Vol. 36, No. 9, pp. 90-102, 1993.

[10] Chandy, K. M., and Kesselman, C., "CC++: A Declarative Concurrent Object-Oriented Programming Notation," Research Directions in Concurrent Object-Oriented Programming ed. G. Agha, P. Wegner, and A. Yonezawa, The MIT Press, Cambridge, MA, pp. 281-313, 1993.

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

[12] Chin, R. S., and Chanson, S.T., "Distributed Object-Based Programming Systems," ACM Computing Surveys, Vol. 23, No. 1, March, pp. 91-124, 1991.

[13] Colin, A., Object-Oriented Reuse, Concurrency and Distribution -- An Ada Based Approach, ACM Press, Reading, MA, 1991.

[14] Cox, B. J., Object-Oriented Programming: An Evolutionary Approach, Second Edition, Addison-Wesley, Reading, MA, 1991.

[15] DoD 95., *Ada 95 Reference Manual*, ANSI/IS/IEC-8652:1995, U.S. Government, 1995.

[16] 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.

[17] Frolund, S., "Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages," ECOOP '92 Proceedings, pp.185-196, June 1992.

[18] Gehani, N., Capsules: A Shared Memory Access Mechanism for Concurrent C/C++, AT&T Bell Laboratories, Murray Hill, NJ, 1992.

[19] 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.

[20] Ishikawa, Y., Takuda, H., and Mercer, C., "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints," OOPSLA '90 Proceeding, ACM

SIGPLAN Notices, Vol. 25, No.10, pp. 289-298, 1990.

[21] 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.

[22] Karaorman, M., and Bruno, J., "Introducing Concurrency to a Sequential Language," Communications of the ACM, Vol. 36, No. 9, pp. 103-116, 1993.

[23] Kiczales, G., "Beyond the black box: open implementation",. IEEE Software, pp. 137-142, .1996.

[24] Lin, C.H., Huang, E.H., Elrad, T., "A language adaptation architecture for reflective concurrent systems", Systems Implementation 2000. R.N., ed. Horspool, 1998 IFIP, Chapman & Hall, 1998.

[25] Liskov, B., and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust Distributed Programs," Concurrent Programming ed. N. Gehani, A.D. McGettrick, Addison-Wesley, Reading, MA, pp. 309-338, 1988.

[26] Lohr, K., "Concurrency Annotations for Reusable Software," Communications of the ACM, Vol. 36, No. 9, pp. 81-89,1993.

[27] 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.

[28] McAffer, J., "Meta-level Programming with CodA", Proceedings of the European Conference on Object-Oriented Computing (ECOOP), LNCS 952, pages 190-214, Springer Verlag, Aug., 1995.

[29] Maes, P., "Concepts and Experiments in Computational Reflection," OOPSLA '87 Proceeding, Orlando, pp.147-156, 1987.

[30] 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.

[31] Matsuoka, S., Taura, K., and Yonezawa, A., "Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages," OOPSLA '93 Proceedings, SIGPLAN Notices, Vol. 28, No. 10, pp. 109-126, 1993.

[32] Meyer, B., "Applying Design by Construct," IEEE Computer, Vol. 25, No. 10, pp. 40-52, 1992.

[33] Nakajima, T., Yokote, Y., Tokoro, M., Ochiai, S., and Nagamatsu, T., "Distributed Concurrent Smalltalk," ACM SIGPLAN Notices, Vol. 24, No. 4, pp. 43-45, 1989.

[34] Nierstrasz, O., "Composing Active Objects: The Next 700 Concurrent Object-Oriented Languages," Research Directions in Object-Based Concurrency ed. G. Agha, P. Wegner, A. Yonezawa, MIT Press, Cambridge, MA, pp.151-174, 1993.

[35] 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.

[36] 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.

[37] Taft, S., "Ada 9X: A Technical Summary," Communications of the ACM, Vol. 35, No. 11, pp. 77-82, 1992.

[38] Takashio, K., and Tokoro, M., "DROL: Object-Oriented Programming Language for Distributed Real-Time Systems," OOPSLA '92, ACM SIGPLAN Notices, Vol. 27, No. 10, pp. 256-294, 1992.

[39] Tsichritzis, D.C., and Nierstrasz, O., "Directions in Object-Oriented Research," Object-Oriented Concept, Database and Applications ed. W. Kim and F. Lochovsky, pp. 523-536, ACM Press and Addison-Wesley, Reading, MA, 1989.

[40] Yokote, Y., and Tokoro, M., "Concurrent Programming in Concurrent Smalltalk," Object-Oriented Concurrent Programming ed. A. Yonezawa and M. Tokoro, The MIT Press, Cambridge, MA, pp. 129-159, 1987.

[41] 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.