Parametric Synchronizer :

An Object Behavioral Pattern for Concurrent Programming

Jesús Álvarez-Gómez Julio García-Martín

Universidad Politécnica de Madrid[1].

Abstract

This paper describes the PARAMETRIC SYNCHRONIZER, an object behavioral pattern capable of managing synchronization and scheduling as parameters in the definition of concurrent objects. Both synchronization models and scheduling policies participate in the pattern's structure as first-class components, and they can be defined from the outside by the programmer. Besides, the pattern allows the dynamic change of the scheduling policy without inconsistencies. Similarly to other related proposals, the PARAMETRIC SYNCHRONIZER provides the maximum amount of concurrency, decoupling method synchronization from method invocations and method executions. As a result, the pattern can be seen as a generic framework/skeleton for the study of new synchronization models on concurrent object-oriented programming, specially focused on solving the inheritance anomaly [Yonezawa 93].

Intent

The PARAMETRIC SYNCHRONIZER externalize the synchronization model and the scheduling policy in a concurrent object execution. Both synchronization and scheduling can be defined from the outside by the user as a parameters of the pattern. Additionally, scheduling policy can be changed dynamically during the execution.

Also known as

Active Object, Synchronizer, Customizable Object Synchronization

Motivation

During the last decade, many proposals on concurrent object-oriented programming posed the need of defining better features and more powerful mechanisms to describe the concurrency. As a result of these works, some authors pointed out the difficulties to combine object-orientation and concurrency in a satisfactory way. Due to the presence of synchronization code into the objects implementation (conditioning the execution) a high portion of reused code when using inheritance must be reprogrammed. The existence of this problem, called the *inheritance anomaly* [Papathomas89, Yonezawa93], moved the designers to re-plan the role of synchronization into the concurrent object-oriented languages. As a fact, the occurrence of the inheritance anomaly highly depends on the synchronization schema provided by the languages. In order to overcome this problem, the synchronization model should be established according the following two principles:

1. The languages should provide a way to describe how to accept requests, that cannot

[1] LSIIS Department, Facultad de informática, Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain. Email: juliog@fi.upm.es

obstruct with concurrently executing methods.

2. Synchronization mechanisms must be a first class mechanisms, so that it should be possible to handle synchronization expressions as any other object related to inheritance (re-use) and message passing.

Nowadays, and thanks to the contribution of many experiences, there exist several synchronization schemas and languages which are able to the deal with the inheritance anomaly. These schemas and languages can be classified into two groups: those based on guarded methods, and those based on accepting set specifications or behaviors set [Papathomas89]. As examples of languages belonging to one of these groups, we can mention the *ABCL/1* language [Yonezawa87], the *Dooji* language [Thomas94] and the *COOL* language [Álvarez97 a, Álvarez97 b]. All these approaches stress the need of separating and locating the synchronization schemas from the bodies of methods, in order to allow fine-grained inheritance/overriding. Moreover, several dynamic operations on the methods must be allowed in order to control which messages are acceptable or not.

Recently, several proposals have been focused on modeling solutions on concurrent object programming by means of design patterns. Examples, such as the ACTIVE OBJECT pattern [Lavender95], the SYNCHRONIZER [Grasso97] or the OBJECT SYNCHRONIZATION PATTERN [Silva96] have shown their ability to de-couple method executions from method invocations in order to exploit a higher degree of parallelism. However, none of these proposals succeed in avoiding the inheritance anomaly, because they do not provide solutions where synchronization the conditions can be specified independently from the method codes.

On its side, the CUSTOMIZABLE OBJECT SYNCHRONIZER pattern [Silva96] manages to allow the complete separation of object synchronization from object concurrency and object functionality. As we see later, this proposal keeps several similarities with the one here presented. As for us, the CUSTOMIZABLE OBJECT SYNCHRONIZER comes up with a balanced distribution of responsibilities among the components defining the pattern's structure. In particular, it proposes the separation between the synchronized access control to the object (not dependent on any synchronization model) from the synchronization model itself (in that case, a predicate based model). Although the solution proposed seems to be very flexible, it has the lack of being highly focused on a particular synchronization model, not bearing other synchronization models in mind. In this situation, we must claim for a more generic synchronization pattern able to support, in advance, whichever model a programmer proposes.

As for the case of languages, patterns based solutions should re-plan their structures in order to integrate synchronization schemas and scheduling policies as first class components. Firstly, because synchronization and scheduling should explicitly participate in the pattern to specify the concurrent execution. Secondly, because they should not be tied to any concrete model (synchronization or scheduling) and, however, take advantage of reusing as much as possible.

The Parametric Synchronizer

For clarity issues, the pattern will be presented in two phases. Firstly, we will present an abstract-level description of the pattern. Under this more abstract description, each component of the pattern is associated to one of the properties addressing the concurrent execution. After that, a more refined version is described. It introduces a more specific pattern relationships (inheritance and delegation) into the components of the former abstract pattern.

Abstract Parametric Synchronizer

As the Figure 1 shows, the structure of the pattern is divided in the following five participants:

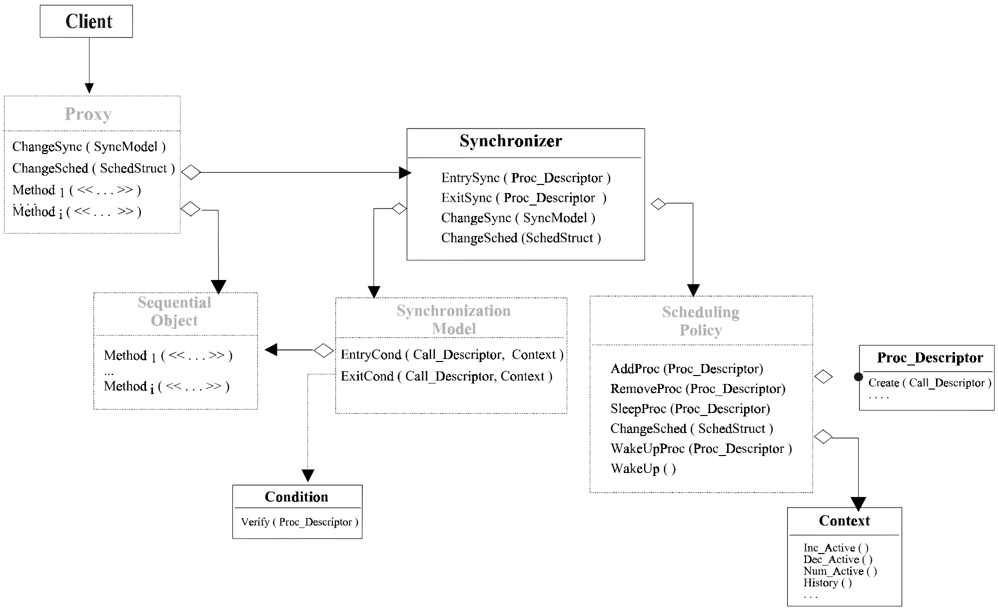

Figure 1. *The Abstract PARAMETRIC SYNCHRONIZER (structure)*

1. **Sequential-Object.** Represents to the sequential object concurrently accessed.
2. **Proxy.** Defines the external interface of concurrency. From the outside, the clients interact on the pattern through the **Proxy** [Gamma95]. It specifies an identical interface as the one provided by the **Sequential-Object**. The Proxy defines two methods (*ChangeSync* and *ChangeSched*) that will be used for dynamic changing of the scheduling policy. Even if these two methods appear on the **Proxy** interface, the clients are not allowed to call them (private access).
3. **Synchronizer.** Behaves as the coordinator for the pattern internal execution. It decides when actions related to other components are carried out. The **Synchronizer** acts as a MEDITATOR pattern of the internal collaborations [Gamma95]. The concrete meaning (the *semantics*) of its operations are fully described now on the operations provided by the **Abstract-Synchorization-Model** and the **Scheduling-Policy**. Because of this circumstance, the pattern avoids that forthcoming concrete synchronization schemas or concrete scheduling policies can change or obstruct **Synchronizer**'s behavior.
4. **Abstract-Synchronization-Model.** Defines the abstract interface representing the synchronization conditions. Such as the **Synchronizer**, it defines two operations related to the access control (the methods *EntryCond* and *ExitCond*). These operations

are responsible of determining if a method requests "*verifies*" the current synchronization conditions, and the signalizing policy (next process to be considered), respectively. In order to do this, the **Abstract-Synchronization-Model** construct a **Condition** from the current state of the synchronization model and the current context conditions. The accounting of active/passive processes currently in the object are stored as a global information by the **Context.** In particular, **Context** keeps track of processes activity into the object, which will be used later to avoid one of the causes of the inheritance anomaly (historical sensitive [Papathomas89]). The **Abstract-Scheduling-Policy** is responsible of managing the **Context**.

5. **Abstract-Scheduling-Policy.** Encapsulates the scheduling policy governing the concurrent execution.

As see now, concurrent executions can be described, abstractly, in terms of collaborations among the above components. Roughly speaking, the execution starts when a client requires from the Proxy the execution of a method *M* with some arguments *X*. The **Proxy** constructs a call descriptor *call (M,X)* and translates it to the **Synchronizer**. At this moment, the **Synchronizer** will evaluate the following sequence of instructions :

```
void M ( Data X )  {
   // Call descriptor construction
    CallDescriptor data = new CallDescriptor ("M");
    data.add( ParameterConversion.convert(X));
   // Synchronization schema
    synchronizer.EntrySync (data);
    object.M (X);
    synchronizer.ExitSync (data);
   }
```

Both the *EntrySync* operation and *ExitSync* operation are always executed in mutual exclusion. The execution of EntrySync itself, will involve to carry out the several actions.

```
public void EntrySync( CallDescriptor data) {
   ProcessDescriptor proc = new ProcessDescriptor( data );

   _entrySemaphore.waitS();
   scheduler.AddProc( proc );
   while (!sync_Schema.EntryCond (data, scheduler.context)
   {
          _entrySemaphore.signalS();
           scheduler.SleepProc( proc );
          _entrySemaphore.waitS();
    }
   _entrySemaphore.signalS();
}
```

First of all, the *EntrySync* constructs a process descriptor (named *proc (M,X))*, and then will order its storage to the **Abstract-Scheduler** (by the execution of *AddProc*). All the requests are stored before they can execute. Now, *EntrySync* must determine whether the request can be executed or not. To do this, the **Synchronizer** tests whether the request fulfils or not the current synchronization conditions. To do this, the **Synchronizer** invokes to *EntryCond ()* on the **Abstract-Synchronization-Model** component. Obviously, this operation encapsulates the action of determining the consistency between the current *synchronization conditions* and *proc(M,X)*. To elaborate an adequate answer, the **Abstract-Synchronization-Model** might require some context information (such as active process for this method, total active process, etc ..). Only the following two answers are possible:

a. In case of *success* (i.e., the answer is *true*), the request is allowed to access and *proc(M,X)* is dispatched to be executed concurrently with the current active processes. On the opposite, the petition to access *fails* and the request must be stored and scheduled (evaluating the operation *SleepProc*). The requests remain stored until a further execution of another request changes the current *synchronization conditions*.

b. When the method code associated to a request has been completely executed, the evaluation of *ExitSync* will notify this end to the **Abstract-Synchronization-Model**. Surely, this last action can change the *synchronization conditions*, and therefore, some requests previously scheduled will have another opportunity to execute.

```
public void ExitSync(CallDescriptor data) {
      ProcessDescriptor proc = new ProcessDescriptor( data );

     _entrySemaphore.waitS();
      scheduler.RemoveProc (proc);
      Condition c = sync_Schema.ExitCond (data, scheduler.context);
      if (_scheduler.context.TotalActive() == 0) {
         scheduler.ChangePlan();
      }
      scheduler.WakeUp (c);
     _entrySemaphore.signalS();
     }
```

The abstract PARAMETRIC SYNCHRONIZER pattern offers a higher-level solution to design concurrent object-oriented applications. On one hand, the pattern seems capable of capturing the features addressing the concurrent executions, with no coupling. On the other hand, the absence of dependencies in the pattern structure highly promotes an independent development of synchronization and methods code, as well as same time avoids obstrusive interactions on a change of any component. The pattern does not solve the inheritance anomaly, because it does not define a concrete synchronization model itself. However, it provides an orthogonal skeleton is well-suited to obtain pattern based solutions capable of overcoming the anomaly. Because of its structure, the pattern induces a more systematic perspective to analyze the problem of concurrency that is being handled.

The next section is devoted to present a more refined version of the abstract pattern proposed now. As strategy, we claim for making each component of the abstract pattern more specific. As we see, the refinement ends by defining another skeleton, over which we will test the success of our pattern to manage several different synchronization schemas (such as *behaviors* [Thomas94, Álvarez97 a] and *logic constraints*) and, likewise, different scheduling policies.

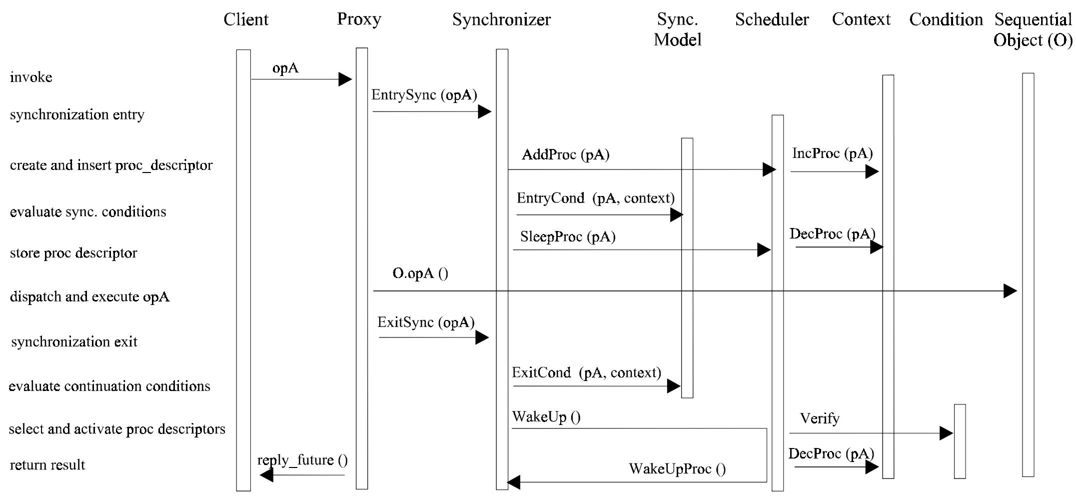

Figure 2. *The Abstract PARAMETRIC SYNCHRONIZER (collaborations)*

Full Parametric Synchronizer

The components of the abstract PARAMETRIC SYNCHRONIZER pattern are refined now in order to achieve a more concise description. Although the new pattern introduces new components it does not modify the former design, remaining the same structure of collaborations. To stress the fine cohabitation between the two levels of the patterns, they appeared together within the new structure (see Figure 3) . The pattern is structured in 4 groups or subpatterns:

1. The **PROXY** group. the **Sequential Object (P)** is linked to a **Proxy (P)** class with identical interface. Concrete Proxies inherit (directly or indirectly) from a common superclass Proxy, which provides the operations to charge both the synchronization model and the scheduling policy. These two operations cannot be redefined in the subclasses.

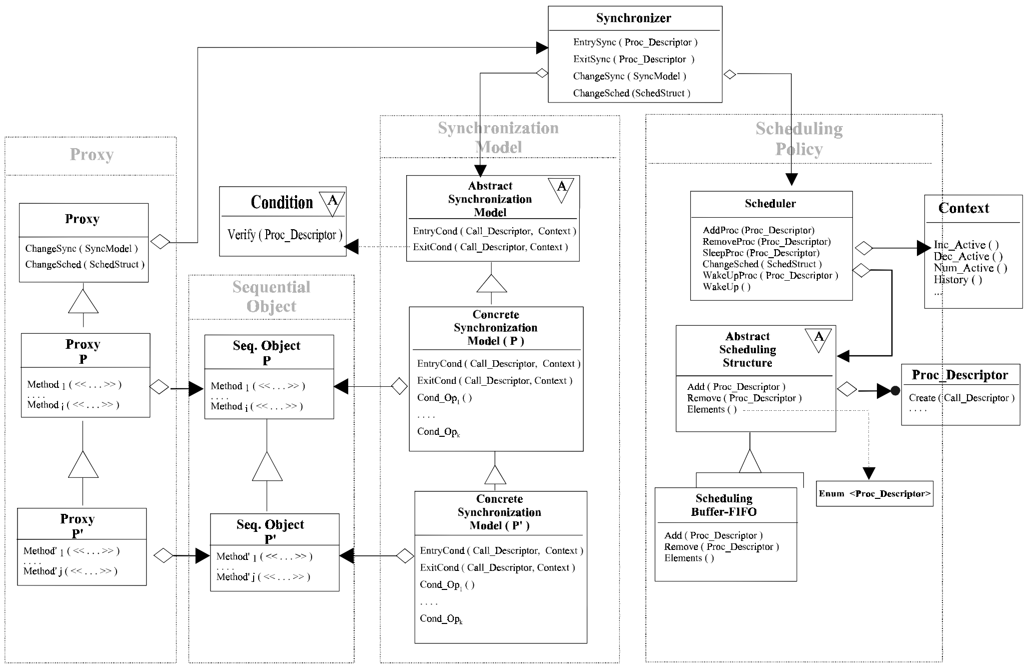

Figure 3. *A refined PARAMETRIC SYNCHRONIZER (structure)*

2. The **SEQUENTIAL-OBJECT** group. Establishing a linear hierarchy of classes, each **Sequential Object** is bounded as an aggregated delegation to its corresponding **Proxy** and **Synchronization Model**.
3. The **SYNCHRONIZATION-MODEL** group. Defines a non linear hierarchy for the **Synchronization Model**s that can be used by the concurrent object. At the top of the hierarchy stays the **Abstract-Synchronization-Model** interface of which must be followed by concrete **Concrete-Synchronization-Models**.
4. The **SCHEDULING** group. It is a pattern itself. On one hand, it defines the **Scheduler** as the manager for the scheduling policies. The **Scheduler** does not provide operations

to manage the process storage or selection. Instead, it uses the set of abstract operations provided by the **Abstract Scheduling Structure**. This structure specifies an abstract collection of process descriptors. It is not tied to any particular structure of representation, which concerns the derived subclasses. Besides, the SCHEDULING group mantains the unique copy of **Context,** which stores a historical track of the execution activity into the concurrent object.

Components of the pattern are completely decoupled. Therefore, specialization or refinement in one group will not provoke obtrusive-effects to the others. This manner, we assure the success of subclassing. Since the **Abstract-Synchronization-Model** does not introduce hard constraints to derivation (only to provide a concrete implementation for the two operations *EntryCond* and *ExitCond),* almost every **Concrete-Synchronization-Model** might fit into the pattern. This way, we are not forcing to concurrent objects to follow any particular synchronization model.

Abstract Scheduling Structure

The initial **Abstract Scheduling Policy** is refined to the following two components: **Scheduler** and **Abstract-Scheduling-Structure**. Similar to the case of abstract data types, **Abstract Structure** uses abstract operations in its definition. In this case, the order relation (the selection) is inferred. This order is used by the **Scheduler** to determine which is the selected request to be considered, among the waiting process. As an advantage, avoiding to fix a concrete representation for scheduling (neither a buffer nor a queue, etc ..) the user can make his best of deciding which is the best representation. In this conditions, a *multi-level scheduling* policies (for *Active / Non-Active* processes as is suggested in [Grasso97]), or common b*uffer-FIFO and buffer-LIFO* policies and more, can be defined as a **Concrete Scheduling Structure**.

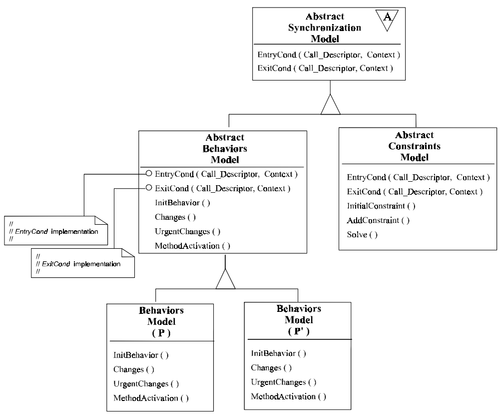

Figure 4. *The SYNCHRONIZATION model hierarchy*

In practice, the superclass **Proxy** makes possible programmer can decide the concrete scheduling policy he wants to follow. In the section of Sample Code is shown an example of **Proxy** construction using these ideas. As a final remark to the section, we recall the ability of **Proxy** to change the scheduling structure at runtime. Different scheduling structure can be interchanged by using a common intermediate structure (in this case, an abstract collection). Whenever a notification for the change of scheduling policy is produced (operation *ChangeSched* in the **Proxy)** the **Scheduler** is ready to carry out the change. The notification of change remains active until the conditions allow it. Even when some other criteria are possible, the **Scheduler** implements the most conservative solution, waiting until there are no active processes executing.

Applicability

Use the PARAMETRIC-SYNCHRONIZER pattern in any of the following situations:

- When it is possible to reuse an object-oriented application in a context of concurrency. Since both sequential and concurrent versions show identical interfaces, concurrent applications could directly work with the new concurrent versions, with no charges.
- When we want to specify the synchronization by means of well-established foundations (formal specifications). The pattern can be formally specified, in the way of [Thomas94].
- When a more refined synchronization schema can increase the degree of parallelism offered by a concurrent object-oriented application.
- When some contextual conditions suggest the change of a synchronization schema (or scheduling policy) to improve the parallelism.
- As a compilation-repository for the development of *true* concurrent object-oriented languages. A language attempting to avoid the inheritance anomaly could use the PARAMETRIC-SYNCHRONIZER as a kernel for the compilation of synchronization and parallelism.
- When the user requires the change of the scheduling policy (both structure or strategy) at runtime.
- When the user decides to define his own scheduling policy (both structure or strategy).
- To define *abstract synchronization schemas.* It is possible to develop an abstract common solution for a family of concurrent problems (readers/writes, producers/consumers), which share a similar synchronization scheme with independence of object implementation.

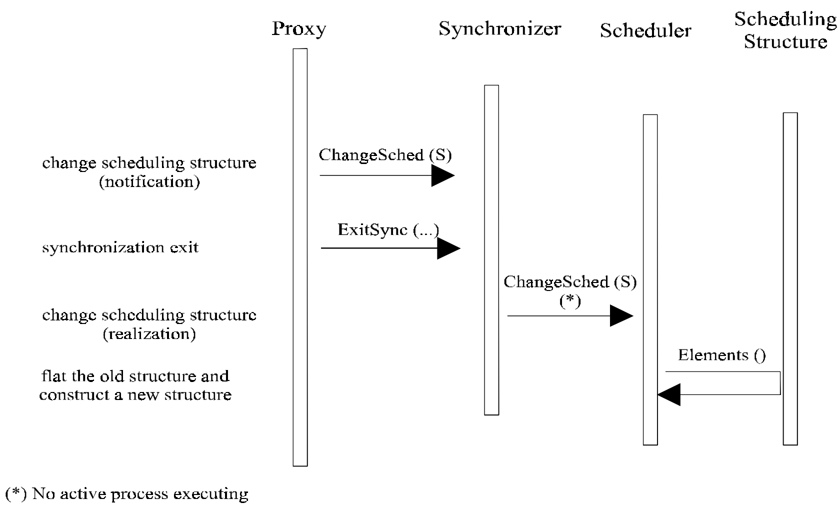

Figure 5. Collaborations on the ch*ange of scheduling policy*

Consequences

The Parametric Synchronizer presents the following advantages:

- Induce a methodology for a most systematic development of concurrent object-oriented applications. By separating objects code from synchronization conditions, we introduce design constraints that force the user to follow a most systematic approach.
- Offer a higher-level degree of encapsulation an re-use with the synchronization code. Synchronization schemas are encapsulated as superclasses.
- Reuse previous designs in new concurrent applications. The PARAMETRIC SYNCHRONIZER admits a direct re-use of sequential classes, scheduling policies and synchronization. In this case, re-use does not introduce any overheads.
- Use a common abstract interface for synchronization allows multiple synchronization schemes. The pattern is particularly well-suited for synchronization schemas based on guarded methods or method sets. We have also tried in the case of *concurrent constraints* schemas.
- Avoid the inheritance anomaly by using a pattern-based schema.
- Introduce dynamic features into the executions and dynamic configuration for synchronization and scheduling.
- Design interface-based PARAMETRIC SYNCHRONIZER schemas to model families of problems with identical synchronization constraints.

Sample Code

The following sample code implementations are written in Java. We present some parts of a program that models the *Producer /Consumer* problem. El problem is modeled using *behaviors* as synchronization model [Thomas94]. We present an implementation that uses as synchronization model behaviors. This model has been already used by the authors as a compilation-Kit for the COOL language [Álvarez97 b].

1. Firstly, we define the superclass **Proxy**. It only defines two operations (ChangeSync and ChangeSched) and a reference to an object **Synchronizer**.

```
public class Proxy {
    public Synchronizer sync;
        public void ChangeSched( Abst_Sched_Struct scheduler) {
            sync.ChangeSched (scheduler);
        }
        public void ChangeSync( Abst_Sync__Model schema )  {
            sync.ChangeSync ( schema );
        }
}
```

2. Now, our implementation must include a **ProxyBuffer** class, the concurrent version from the corresponding sequential bounded buffer. The ProxyBuffer extends to the generic Proxy and inherits all of its functionality. As can see on the code, the schema admits the inclusion of variables *future,* which are used to return results if needed.

```
public class ProxyBuffer extends Proxy  {
  private Buffer buffer;
  private Synchronizer sync;

   ProxyBufferCliente( Abst_Sched_Struct sched , Abst_Sync sync_model, Buffer b) {
        this.buffer = buffer;
        this.sync = new Synchronizer(sched, sync_model);
   }
```

```
    void put( char c)
     ....
    char get( )  {
        CallDescriptor data = new CallDescriptor ("get");
        sync.EntrySync (data);
        char c = buffer.get();
        sync.ExitSync( data );
        return c;
    }
}
```

3. The implementation of the **Abstract Scheduling Structure** can be easily carried out by using Java interfaces, a well-suited mechanism to describe abstract classes.

```
interface Abst_Sched_Struct {
      void Add ( ProcessDescriptor p );
      void Remove ( ProcessDescriptor p );
      Enumeration Elements ();
}
```

4. Finally, we have to establish the synchronization schema for the Producers/Consumers problem. As said before, we propose a behaviors-based solution. We define the abstract class *BehCondition* related to the behavior model.

```
public class BehCondition implements Condition {
    BehaviourModel model;
    CallDescriptor call;

    BehCondition (BehaviourModel model, CallDescriptor call) {
        this.model = model;
        this.call = call;
    }
    public boolean Verify () {
       model._VerifyBehavior (call);
    }
  }
```

5. As see below, the operations *EntryCond* and *ExitCond* are abstractly described by the Abst_Sync_Model, and it keeps a reference to the global component **Context**.

```
public abstract class Abst_Sync_Model  {
      protected Context context;
      abstract public EntryCond ( CallDescriptor call );
      abstract public ExitCond ( CallDescriptor call );
      public SetContext (Context c) {
          context = c;
      }
}
```

6. Now, the operations *EntryCond* and *ExitCond* are concreted in each synchronization model (in this case, bahaviors model), as follows:

```
public abstract class BehaviorModel implements Abst_Sync_Model  {
      Behavior state;
      abstract public void            _InitBehaviors();
      abstract public BehCondition    _Changes( CallDescriptor c);
      abstract public void            _UrgentChanges( CallDescriptor c);
      abstract public boolean         _MethodActivation( CallDescriptor c);
      abstract public boolean         _VerifyBehavior ( CallDescriptor c );


      public boolean EntryCond (CallDescriptor call)     {
            BehCondition cond = new BehCondition (this, call);
            return cond.Verify () &&  state.MethodActivation (call, context);
      }
      public Condition ExitCond (CallDescriptor call) {
            return _Changes( call, context );
      }
}
```

At this case, the evaluation of the *EntryCond* operation will test if the request *call* satisfies the current behavior conditions under some given current contextual conditions *c.* Additionally, it must be tested if the activation conditions for the method are satisfied as well.

Bibliography

[Alvarez97 a] Jesús. Álvarez-Gómez. *Concurrent Object-Oriented Programmirog Languages– The COOL language.* PhD. Curses. Facultad de Informática. Universidad Politécnica de Madrid. June, 1997.

[Alvarez 97 b] Jesus Álvarez, Julio García. *COOL : A Concurrent Object-Oriented Language*. III Jornadas sobre Tecnología de Objetos, ATI, Sevilla. October, 1997.

[Grasso 97] E. Grasso (1997). *Synchronizer – An object Behavioral Pattern for Concurrent Programming*. EuroPloP'97. 1997

[Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns. Elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[Lavender 95] R. Lavender, D. Schmidt. *Active Object*. Conf. Pattern Languages of Programming, Monticello, Illinois, September 1995.

[Nierstrasz 93] O. Nierstrasz. *Composing Active Objects.* Research directions in concurrent object-oriented programming. The MIT Press, Cambridge, Massachussets, London 1993.

[Papathomas 89] M. Papathomas. *Concurrency Issues in Object-Oriented Programming Languages*. Geneve University, 1989.

[Silva 96] A. R. Silva, J. Pereira and J. A. Marques. *Object Synchronization Pattern*. 1996.

[Thomas 94] L. Thomas. *An object-oriented concurrent language for reusability and reuse of synchronization components*. Tokyo University, 1994.

[Yonezawa 87] A. Yonezawa, J. Briot, and E. Shibayama. *Object-Oriented Concurrent Programming in ABCL/1.* Proc. of OOPSLA'87. ACM Press, September 87.

[Yonezawa 93] A. Yonezawa, S. Matsuka. *Analysis of inheritance anomaly in object-oriented Concurrent Programming Languages*. Research directions in concurrent object-oriented. The MIT Press, Cambridge, Massachussets, 1993.