The Adaptive Arena:
A Concurrent Object-Oriented Model

Tzilla Elrad
cselrad@mina.cns.iit.edu
Atef Bader
abader@lucent.com

Abstract

Most of the current concurrent object-oriented approaches do not address the issue of separation of concern between synchronization and scheduling controls inside the concurrent objects. This paper presents a concurrent object-oriented model in which a concurrent object, which represents a shared resource abstraction in our model, is decomposed into a hierarchy of abstractions: a shared data abstraction, a synchronization abstraction, and a scheduling abstraction. It will be shown that the separation of concern among the three major components of the concurrent objects avoids many of the conceptual difficulties that arise when integrating concurrency into object-oriented paradigm. Our model provides explicit, declarative, and reusable first class components for synchronization and scheduling controls as it has been the case for data and operations in the sequential object-oriented languages. The notion of scheduling policy inheritance in our model facilitates the process of engineering adaptability in the development of the intelligent reactive/adaptive systems.

keywords: Concurrent object-oriented programming, Soft-real-time/Adaptive systems, Reuse, Synchronization constraints, Scheduling protocols, Adaptive-Arena.

1. Introduction

Building a concurrent application requires the programmer to specify the synchronization and scheduling constraints of the concurrent actions and in order to specify these constraints the programmer has to be provided with constructs that will enable him to express these constraints. Concurrent programming languages that provide these constructs can be classified as either procedural, like Concurrent C[21], SR[5], Ada83[13], or object-oriented, like Composition-Filters [2], Capsule[22], DRAGOON [6], and μC++ [10]. Using the object-oriented paradigm [9, 20, 26, 30, 34] to build the concurrent applications is preferable to the procedural model since it allows code reuse and programming by extension, and enhances the management of large-scale software projects.

The Adaptive-Arena is a concurrent object-oriented language advocates the separation of synchronization and scheduling controls from the method bodies in order to regulate the intra-object concurrency as well as promote code reusability. Therefore, concurrency control as well as the method bodies within the concurrent objects can be reused, extended, or overridden. The Adaptive-Arena facilitates the design of extensible libraries of concurrent components as well as the development of the design patterns for a wide range of concurrency problems. The intra-object concurrency is controlled through local synchronization and scheduling decisions for each object.

In the next section we briefly overview the existing approaches for integrating concurrency with the object-oriented paradigm. In the subsequent sections, we present the Adaptive-Arena model along with its solutions for a wide range of concurrency problems. Next, The Adaptive-Arena model is compared with the related work and finally the paper’s contributions are summarized.

2. Concurrent Object-Oriented Programming

Several of the concurrent programming languages have used the notion of encapsulated object in their programming models. For example, Ada95[14] uses a server-based approach based on the rendezvous mechanism, Monitors[25] uses an abstract data type mechanism, Concurrent C[21], SR[5], and [4, 11, 24] are based on the abstract data type with path expressions, approaches that are based on the sequential objects like [3, 10, 22, 28], and actor-based approaches[1].

Synchronization and scheduling controls form the major components of concurrent systems and a number of models [12, 19, 23, 28, 2] have been proposed in order to employ the object-oriented paradigm in building these systems. The integration of concurrency and object-oriented technology raised many problems that makes it difficult to employ the object-oriented technology in building the concurrent systems in a straightforward manner. Therefore, the initial attempts [3, 10, 12, 19, 22, 23, 28] were made to separate the synchronization code from the functional code. The separation of concern raised another problem that is the difficulty of reusing the synchronization code. The tight coupling between the synchronization code and the functional code causes limitations on code reuse and programming by extension. Synchronization constraints often conflict with inheritance and that what has been classified by [28] as the *inheritance anomaly* problem where the addition of a method in the subclass requires modifications to the synchronization constraints of methods in the superclass. The distinguishing between synchronization and scheduling controls was first presented in [15, 16] where concurrent systems are partitioned into two major components: concurrency control and functionality control. Capsules[22] and Ada95[14] addressed the issue of separation of control between synchronization and scheduling controls but did not completely utilize its concepts. Like synchronization control, scheduling control may cause another potential problem when employing the object-oriented technology in building the concurrent object-oriented systems and that is due to the interference between scheduling and inheritance. Scheduling should be represented in term of first class components in order to enhance code reusability. Scheduling policies should not cause limitations on code reuse; rather language constructs should be provided in order to allow reuse of these scheduling policies. The isolation of the scheduling policy from the synchronization code and functional code will enhance code reusability and eliminate the interference between scheduling and inheritance.

It has been stated in [7] that the hierarchical representation of the software system has a major impact on the design and development of reusable components. Therefore, our initial attempt was to focus on aspects that are fundamental to representation of application entities. We partition the concurrent object into a hierarchy of abstractions that will aid in the design and the development of the concurrent applications. Adaptability [16, 17] is an important factor that enables complex systems to evolve in order to meet future needs. Therefore, our approach identifies the methodology into which design patterns [17, 20, 32] for a wide range of concurrency problems, similar to the design patterns presented in the ACE framework [32], can be easily identified and implemented.

In this paper, we examine the manner in which changes in the components of an object of a concurrent class affect other components of the same object. The focus here is to partition the concurrent system into a set of components, synchronization, scheduling, and functionality controls, in order to support inheritance for the these components, and minimize changes that are due to the interaction between these components if complete reusability can not be achieved. The separation of concerns is handled within our framework by defining different constructs for synchronization and scheduling controls. Method invocations within the object can be serialized or concurrent based on the specification of scheduling and synchronization constraints at the object level.

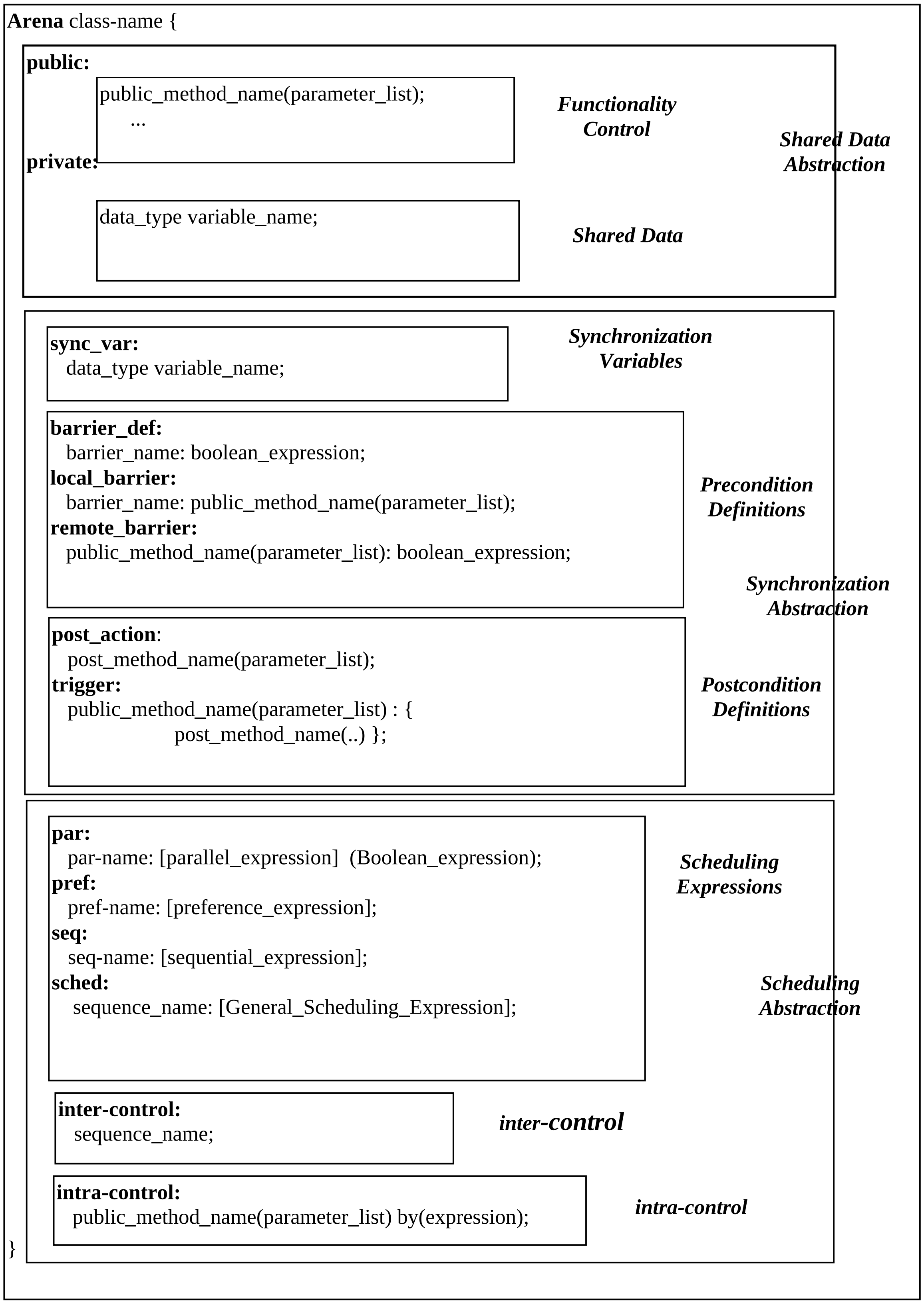
The primary goal of our model is to integrate concurrency into the object-oriented paradigm [9, 26, 30] and introduce a new notion that we call synchronization and scheduling abstractions. In our framework, we define abstraction based on the definitions given in [7, 9, 33] as being a mechanism that provides the conceptual boundaries that will aid the developer in building a hierarchical view of the concurrent system components. Scheduling and synchronization abstractions are expressible as first class components. By giving abstract names for the synchronization and scheduling expressions we can reuse and extend those expressions in subclasses. *Scheduling abstractions* in our framework are used in order to implement different scheduling protocols. These scheduling abstractions hide the underlying implementation of these scheduling protocols. Our scheduling abstractions will be used to implement the *inter-invocation* and *intra-invocation* scheduling policies. In our model, we defined a new user-defined type that we call *arena*, which has similar capabilities to C++ [34] classes except for the new features that we added in order to support reuse for the concurrent applications and regulate the intra-object concurrency. Through the use of synchronization and scheduling abstractions, the behavior of an object can be reused, specialized, or extended.

3. Adaptive Arena Model

The *Adaptive-Aren*a, which represents a shared resource abstraction, is conceptually divided into three major components: *shared data abstraction*, *synchronization abstraction*, and *scheduling abstraction*. The *shared data abstraction* is composed of the *shared data* and the *functionality control* that has similar concepts to sequential objects. The *functionality control* of the *shared data abstraction* will consist of a set of member functions that will access the shared data. Synchronization and scheduling abstractions control the access of the member functions to the shared data. Figure 1 shows the general specification of the Adaptive-Arena class in our framework.

3.1 Synchronization Abstraction

Synchronization controls are the decision controls that enable or disable a nondeterministic method invocation for selection and they are implemented using guards and barriers in our model. In our framework, synchronization abstraction is composed of three components *synchronization variables*, *precondition definitions*, and *postcondition definitions*. *Synchronization variables* are the set of the variables used in constructing the synchronization expressions that are given abstract names. *Synchronization variables* can be declared using the *sync_var* language construct. The *precondition definitions* block is composed of three components: *barrier_def* which is used to give abstract names, barrier names refer to the internal state of an object, for the synchronization expressions, *local_barrier* which is used to associate barrier names with the member functions, and *remote_barrier*. In [15], *remote_barrier* is called *mutual control*, which is used to associate Boolean expressions, which refer to the internal state of an object along with the caller parameters, with member functions. The *postcondition definitions* block is composed of two components: *post_action*, which is used to declare the postaction functions that are used to modify the synchronization variables, and *triggers*, which are used to create an association between the postaction functions and the member functions which operate on the shared data; Only a post function that has been defined in the *post_action* block may have write access to the synchronization variables; Other member functions of the adaptive arena have read only access to the synchronization variables. The local and remote components represent the named guards or barriers that are associated with methods rather than sets. Postactions are executed in mutual execution since a



```
Arena class-name {
  public:
        public_method_name(parameter_list);
             ...
  private:
        data_type variable_name;

  sync_var:
     data_type variable_name;
  barrier_def:
     barrier_name: boolean_expression;
  local_barrier:
     barrier_name: public_method_name(parameter_list);
  remote_barrier:
     public_method_name(parameter_list): boolean_expression;
  post_action:
     post_method_name(parameter_list);
  trigger:
     public_method_name(parameter_list) : {
                     post_method_name(..) };

  par:
     par-name: [parallel_expression]  (Boolean_expression);
  pref:
     pref-name: [preference_expression];
  seq:
     seq-name: [sequential_expression];
  sched:
      sequence_name: [General_Scheduling_Expression];
  inter-control:
     sequence_name;
  intra-control:
     public_method_name(parameter_list) by(expression);
}
```

Figure 1. The Adaptive-Arena Class Representation.

postaction may reference a synchronization variable that has been used in more than one barrier. A modification of a synchronization variable that has been defined in the *sync_var* block by a non postaction method is detected by the translator and reported as an error.

3.2 Scheduling Abstraction

During the execution of a concurrent program, threads race to be scheduled. The order of the selection of these schedulable entities can be specified by scheduling expressions [4, 11, 24] and these scheduling expressions are called scheduling policies or scheduling protocols. Scheduling protocols form a major component of the concurrent systems. The interaction and ordering of method executions in our model within the concurrent object can be specified by expressions that have similar capabilities to Path Expressions[4, 11, 24]. In order to support the addition of new scheduling policies for the newly added methods and to allow the extension of the scheduling policies for the existing methods within the concurrent object, we created what is called *Scheduling abstraction,* which allows the specification of the scheduling protocols in isolation of the sequential code for the method bodies. *Scheduling abstraction* in our model has been addressed at two levels: *inter-invocation control,* which is used to specify the order of execution for a set of method invocations, and *intra-invocation control,* which is used to specify the order of servicing pending requests in the method queue.

3.2.1 Inter-invocation Control

Scheduling abstraction at the inter-invocation has been addressed in our framework at three dimensions: *parallel, sequential,* and *preference*. We created a construct for each of these scheduling components. Abstract names have been given to the parallel, sequential, and preference scheduling expressions in order to allow reusability of these scheduling expressions in the derived classes. The *inter-invocation control* in our proposed model is composed of five components: *par*, *pref, seq, sched*, and *inter-control*. Through the use of the *par* construct we can identify the set of methods that may run in parallel and give that set an abstract name that can be reused in the derived classes in order to create an extended set. The *pref* and the *seq* constructs are used to name and create different scheduling subexpressions with different properties that will be discussed shortly. The *sched* construct is used to define the scheduling policy of the *adaptive arena* by combining the subexpressions that have been defined in the *pref* and the *seq* blocks. Using the *inter-control* construct we can identify the scheduling policy that should be applied during runtime. A concurrent adaptive object may choose to switch from one scheduling policy into another by using on the fly analysis. Through the use of this construct we can create an intelligent adaptive object. By giving *abstract* names for these scheduling expressions, we enhance code reusability for these scheduling expressions.

These scheduling abstract names can be inherited in order to define new scheduling abstractions for inherited and new methods, or a new scheduling abstraction can be defined for the inherited and new methods. By doing so, we support code reusability at two levels: functional level and behavioral level. Each of the behavioral components can be defined, inherited, or overridden. Several access protocols can be expressed and implemented easily using our scheduling constructs.

To construct different scheduling policies that will emulate the interaction between the different competing methods within the concurrent object and to specify the order of their execution we designed five scheduling operators that will aid in the specification and implementation of these scheduling policies. The following sections will review in detail these scheduling operators.

3.2.1.1 Nondeterministic Selection (+)

This operator is used to simulate the nondeterministic behavior for a set of open alternatives in the concurrent application. The execution for the methods in the sequence that has been specified will be carried out in some arbitrary order. For example the expression s_0: [$m_i + m_j$] is a legal sequence where s_0: is the sequence name. If this operator is used to separate sequences rather than method names, then the same semantics will be held. The nondeterministic operator, +, is used to separate a list of schedulable entities. There is no guarantee in the order of execution for these entities. Rather we assume fairness and eventually each of these open alternatives will be selected for execution. The feasibility of the fairness notion is supported through the use of random assignments to priority variables [18, 31].

3.2.1.2 Sequencing Operator (;)

This operator is used to enforce the order of the execution for a set of methods in the order that has been specified. For example, s_0: [m_i; m_j] is a legal expression in our model. After the completion of method m_i, the only method that may start executing next is method m_j and if there are any another pending invocations for another methods then these invocations will be delayed until the acceptance and completion of an invocation of method m_j. If the *inter-control* construct of an object has selected this sequence for scheduling then the only acceptable invocation will be an invocation for method m_i. The sequential operator " ; " is used to impose a strict sequence order on a list of schedulable entities. For example, in the sequence $M_i;M_j$, method M_j can be started only if the last terminated method is M_i , that is to say, method M_i upon termination will set the identity of the next schedulable method which is M_j. In general, if the strict sequence expression has the following format $M_1;M_2; ... ;M_n$ then all of these methods will be executed in a mutual exclusion and once method M_i terminates then it will set the identify of the next schedulable method, which is method M_j, where $j=i+1,\ 1 \le i \le n - 1,$ and $2 \le j \le n.$ If the strict sequence expression is composed of a set of sequence names, like $S_1;S_2; ... ;S_n$, rather than method names, then the first method that appears at the beginning of sequence S_j is allowed to start its execution once the rightmost method in sequences S_i finishes its execution where $j=i+1,\ 1 \le i \le n - 1,$ and $2 \le j \le n.$.

3.2.1.3 Preference Operator (>)

This operator can be used to prioritize the alternatives easily and incrementally reuse the scheduling behavior of a superclass. The prioritized entries of the soft-real-time applications can be easily implemented through the use of this operator. The following expression s_0: [$m_i > m_j$] is a legal expression in our model. s_0 is the name of the sequence and the meaning of this sequence is as follows: the execution of m_i is always preferred over method m_j; even if there is more than one pending invocation for m_j. The invocations of m_j will be given the opportunity to execute only if there is no pending invocation for m_i. Method m_j may be preceded by a sequence rather than a single method and in this case the preference will be given to a set of methods rather than a single method as in the following example s_1: [$m_k > m_l$] , s_2: [$s_1 > m_j$]. The invocation of method m_j can be started only if there is no pending invocation in the sequence s_1.

3.2.1.4 Alternation Operator (>>)

The alternation operator is used to keep alternating the execution of two method or groups of methods. The following expression s_0: [$m_i >> m_j$] is a legal expression in our model. s_0 is the name of the sequence and the meaning of this sequence is as follows: if there is a pending invocation for method m_i and another pending invocation for method m_j , then preference is given to the execution of one invocation for method

m_i followed by the execution of one invocation for method m_j. If there is more than one pending invocation for each of these methods then the concurrent object will keep repeating the above protocol as long as there are pending invocations for these methods. If m_j is preceded by a sequence rather than a single operation as in the following example s_1: [$m_k >> m_l$] , s_2: [$s_1 >> m_j$] , then a pending invocation for method m_j should be executed after scanning the pending invocations of the methods in the sequence s_1 and executing at most one pending invocation in the sequence s_1. The alternation operator can be used to keep alternating between two method groups and it has the following format $S_1 >> S_2$. If there is more than one pending invocation that belong to methods in one of the two groups S_1, *and* S_2 , then each time a method call from each group will be selected for execution. If there is no pending invocation which belong to group S_2 then method calls which belong to S_1 will be allowed to start their execution as long as there is no pending invocation which belong to a method in group S_2. This operator can be used to implement the protocol *Multiple Readers or One Writer with Writers Having a Higher Priority & no Indefinite Waiting for Readers* [22].

3.2.1.5 Strict-precedence Operator (<<)

The strict-precedence operator is used to test the identity of the last terminated method. The general format of a scheduling expression that uses this operator is *(list of methods) << sequence* , and the semantic of this expression is as follows: the first schedulable method in the set *sequence* is allowed to start executing if the last terminated method belongs to the set (*list of methods)*; If the scheduling expression has the following format: ~*(list of methods) << sequence* then the meaning of this expression is as follows: the first method in *sequence* is allowed to start executing if the last terminated method does not belong to the set *(list of methods)*.

3.2.2 Intra-invocation Control

Request Parameters is one of the categories that has been identified by Bloom [8] for expressing synchronization constraints. A server object should be able to access the parameters of the request, method invocation, in order to accept or block a request of a client. This category has been defined by Elrad [15] as *mutual control*. Capsules [22] supports this category through the use of the *suchthat* construct , and Ada [13, 14] provides the *requeue* statement in order to simulate it. Each method in the synchronized object is associated with a queue for queuing the incoming requests if they can not be serviced immediately. The order of servicing these requests is defined by [15] as *forerunner control*. These queued requests can be serviced by the FIFO policy or the priority of the caller in Ada. In Capsules, the programmer can define his own scheduling policy for servicing these pending calls through the use of the *by* clause. In our model, we provide the *by* construct, which has similar capabilities to the one that presented in [22]. Through the use of the *by* construct, we have a complete control over the order of execution of pending requests in the method queues.

3.2.3 Evaluating the Scheduling Expressions

Methods within the Adaptive Arena are, by default, executed in mutual exclusion. In order for a method invocation to start its execution it should satisfy the synchronization constraints in the local and remote parts, and then satisfy the scheduling restrictions. If there is no barrier associated with a certain method in the remote or local specification parts then that method will be executed if the current scheduling restrictions permit to do so. Each method is associated with a queue in order to queue the method invocation calls. These invocation calls, by default , are executed in a FIFO order. A method call to the Adaptive Arena is allowed to execute if after satisfying the scheduling sequence conditions the mutual exclusion condition is satisfied or if the method call is allowed to execute in parallel with the currently executing method.

3.3 The Adaptive-Arena Solution

The employment of the object-oriented paradigm in the development of concurrent systems raised many problems that are due to inheritance and the lack of expressive constructs for the development of these systems. None of previous approaches[1, 3, 10, 12, 19, 22, 23, 27, 28] has addressed the distinction between synchronization and scheduling controls. Our approach promotes reuse of the synchronization constraints and the scheduling protocols of the concurrent objects.

In the Adaptive Arena model, we consider the thread to be the basic unit of concurrency. The *adaptive arena* is a passive object. Threads access the adaptive arena passive object based on the scheduling table that has been constructed by the runtime system of the adaptive arena model. In reality, the code generator creates the basic entries of that table and the runtime unit puts those entries together to formulate the scheduling table that will be consulted by each thread upon accessing the adaptive arena passive object.

Once a thread completes execution within the adaptive arena, it updates its corresponding entries in the scheduling table. In this way, threads coordinate among each other through the lifetime of the passive object. In the following two sections we present our solutions to a wide range of concurrency problems that arise during the development of concurrent object-oriented systems.

3.3.1 Inter-invocation Control

In our model, we provide a number of constructs for controlling the order of the execution of different methods within the concurrent object. To demonstrate the expressiveness of these constructs, we applied them on a number of concurrency problems. Figure 2 shows the representation of the bounded buffer class in our model. Figure 3 shows how our approach prevents the occurrence of the *history-only sensitiveness* anomaly [28]. In our solution the addition of method gget(), that may execute only if the last terminated operation is not a put operation, does not require modification of the superclass. Separating the scheduling protocol from the synchronization code, clearly enhances code reusability and minimizes code modifications for the sequential and synchronization code. A scheduling protocol of a superclass can be wrapped by a new scheduling protocol in the subclass. An *abstract mixin* class, like the Lock class, can be easily mixed with another class in our framework. The capability of locking an object can be accomplished by inheriting the Lock class and the superclass and then define the scheduling protocol in the subclass. Figure 4 shows our solution to the *state modification anomaly* [28].

The main concern in the reader/writer protocol [6, 14, 22] is to ensure the integrity of the shared data and the avoidance of starvation for readers or writers. Previous approaches used the Boolean variables within the method bodies in order to avoid starvation and modify the superclass in order to enforce it. On the other hand, our approach is simple, clean, and starvation free. In our approach, there is no need to modify the sequential code of the superclass. The only necessity is to inherit and extend the scheduling behavior. Figure 5 shows the representation in our model for the reader/writer protocol with priority for writers. Preference is always given to writers; readers may starve. In order to implement the fair reader-writer protocol in other approaches [6, 14, 22], method bodies should be modified in order to include a Boolean variable that enforces the alternation and that what we try to avoid always; A subclass should not modify the sequential code of a superclass. On the other hand, our approach does not modify the method bodies of a superclass. Rather it inherits the superclass and then define the new scheduling protocol without affecting any of the method bodies in the superclass. Figure 6 shows the representation of the fair reader-writer protocol in our scheme after extending the class that has been presented in Figure 5.

Scheduling Wrappers are used when introducing new methods on the subclass and defining a new scheduling protocol on top of the superclass scheduling protocol. For example, suppose we want to define the new operation ReadNoItems, which will return the number of items that are currently in the buffer using the bounded buffer class that has been presented in Figure 2. Unlike other approaches [3, 10, 12, 19, 22, 23, 28, 29], the scheduling protocol along with the sequential code of the superclass need not be modified in the subclass. Figure 7 shows the implementation of the extended buffer in our scheme where a new method has been added along with a new scheduling protocol. In our model, different components can be inherited and glued together in order to allow the incremental evolution of the software system.

Sometimes there is a need to have an explicit preference control over the order of invocation of competing entries especially in soft-real-time applications. Therefore, a mechanism should be provided in order to enable him to control the dispatching order. Ada95 [14] and Capsules[22] have been designed to support building the soft-real-time applications. Capsules provides the c_waiting construct and Ada provides a family of entries along with the ’COUNT construct in order to prioritize a set of entries. For example, suppose that there are two services that a superclass provides; Service Medium and Service Low where Medium has an absolute preference over Low. A subclass may inherit the superclass and add a new service High that has a higher preference over the services of the superclass. Unlike other approaches, in our approach the addition of the new service High will not affect the preferences in the superclass. Rather, the subclass will use the scheduling preference of the superclass in order to implement the new scheduling preference in the subclass. Figure 8 shows the representation of the prioritized entries in our scheme, and Figure 9 shows the extended prioritized entry class.

In order to build an intelligent adaptive object, a mechanism ought to be provided that will aid the object in selecting the right scheduling policy by performing on the fly analysis during runtime. Concurrent real-time systems will greatly benefit from this capability in our framework. Through the use of the inter-control construct, we offer the developer the opportunity to engineer adaptability within the synchronized object, in order to fine-tune their reactions during runtime. For example, in the bounded buffer example there is a need to design an adaptive scheduling protocol that prefers put over get if the buffer is less than half-full and get over put if the buffer is more than half-full. Figure 10 presents an example of an intelligent adaptive buffer in the Adaptive-Arena model.

3.3.2 Intra-invocation Control

Each method in the Adaptive-Arena objects is associated with a queue for queuing the incoming requests if they can not be serviced immediately. In our model, these queued requests are serviced on a FIFO basis. If there is a desire to alter this policy, then the developer can do so through the use of the *by* construct in order to specify the customized servicing policy. For example, Figure 11 shows an extended version of the bounded buffer where requests will be serviced based on the size of the request.

```
Arena buffer {
sync_var:
  int noitems, in, out;
barrier_def:
  NotFull :  (noitems < MaxSize);
  NotEmpty : (noitems > 0) ;
local_barrier:
  NotFull: put(int item);
  NotEmpty : get(void);
pref:
  s0: [ put (int) + get(void) ];
sched:
  s1: [ s0 ];
inter-control:
   s1;
post:
 Inc(int *value);
 Dec(int *value);
trigger:
 int put(int item): { Inc(&in); Inc(&noitems); }
 void get(void): { Inc(&out); Dec(&noitems); }
}
buffer::buffer(int size){
        max = size;
        buf = new int(max);
        int = out = 0;
      }
buffer::Inc(int *value){
      ++(*value);
      }
buffer::Dec(int *value){
      --(*value);
      }
buffer::put(x: in integer){
        buf[in] := x;
      }
int buffer::get(void )  {
        x := buf[out];
        return x;
      }
```

Figure 2. Bounded Buffer Class Representation in the Adaptive Arena

```
Arena H-buffer: public buffer {
public:
     int gget(void);
local_barrier:
  NotFull: gget(void);
sched:
  s2: [ s1 +  ~(put(int)) << gget(void) ];
inter-control:
  s2;
};
H-buffer::H-buffer(int size):
           buffer(int size) ;
int H-buffer::gget(void )  {
  x := buf[out];
  return x;
}
```

Figure 3. Extended Bounded Buffer in the Adaptive Arena Representation

```
Arena Lock:{
  int locked;
seq:
 s2: [ lock(void) ; unlock(void) ];
sched:
  s3:[s2];
inter-control:
  s3;
public:
  void lock(void)  { };
  void unlock(void) { };
}
Arena lb-buf: public buffer, Lock {
sched:
  s4: [ s1 + s3 ];
inter-control:
  s4;
lb-buf(int size) : buffer(int size) { }
}
```

Figure 4. Adaptive Arena Solution for the State Modification Anomaly

```
Arena rw {
  int I;
pref:
 s1: [ write(int) > read(void) ];
par:
  par1: [ read(void)# ];
sched:
  s2: [s1];
inter-control:
  s2;
public:
  int read(void);
  void write(int x) ;
};
```

Figure 5. Reader/Writer Protocol in the Adaptive Arena Representation

```
Arena Fair_rw : public rw {
pref:
  s2: [ write(int)  >> read(void) ];
sched:
  s3: [ s2];
inter-control:
  s3;
}
```

Figure 6. Fair Reader/Writer Protocol in the Adaptive Arena representation

```
Arena RW-buffer: public buffer {

pref:
  s2: [ s1 >> ReadNoItems(void) ];
par:
  par1:[ReadNoItems(void)#];
sched:
  s3:[s2];
inter-control:
  s3;
public:
RW_buffer(int size) : buffer(int size) { }
int ReadNoItems(void);
}
int RW_buffer::ReadNoItems(void){
      return(noitems) ;
}
```

Figure 7. Scheduling Wrappers in the Adaptive Arena Representation

```
Arena Service_A {
public:
  Service_A(void);
  void Medium(void);
  void Low(int item) ;
pref:
 s1: [ Medium(void) > Low(int) ];
sched:
 s2: [s1];
inter-control:
 s2;
}
```

Figure 8. Prioritized Entries in Adaptive Arena

```
Arena Service_B: public Service_A {
public:
  Service_B(void);
  void High(void);
pref:
  s3: [ High(void)  > s2 ];
sched:
  s4: [s3];
inter-control:
  s4;
}
```

Figure 9. Extended Prioritized Entries in the Adaptive Arena

```
Arena  adaptive_buffer: public buffer {
pref:
  s2: [ put() > get()];
  s3: [get() > put()];
sched:
  s4: [s2];
  s5: [s3];
inter-control:
  if(noitems > (Maxsize/2)) s4
 else s5;
}
```

Figure 10. Adaptive Buffer Representation in the Adaptive Arena

```
Arena buffer {
remote:
  put(int *item, int requestSize)  :
     (requestSize <= (Max - noitems));
  get(int requestSize) :
     (requestSize <= noitems);
sched:
  s1: [ put(int *, int)  + get(int) ];
inter-control:
  s1;
intra-control:
  put(int* item, requestSize)
          by(requestSize);
 get( requestSize) by(requestSize);
}
```

Figure 11. Extended-Bounded Buffer in the Adaptive Arena

4. Related Work

The separation of the synchronization code from the functional code in the concurrent object-oriented paradigm has been addressed by many proposals [2, 3, 6, 8, 10, 14, 22, 23, 27, 29], but none of these proposals has addressed the issue of separation between synchronization and scheduling controls. In Composition-filters [2], synchronization constraints can be specified through the use of filters of class *Wait;* reuse of the synchronization code is very restrictive and the specification of different coordination scenarios is not possible in many cases. Capsules[22] suffers from the inheritance anomaly problem and the modification of the method bodies to regulate the intra-object concurrency is a must in many cases. In [8], the use of negative guards has been employed to promote reuse of the concurrent objects but this approach offers a very limited degree of code reusability.

Models that are based on the acceptance sets like the one proposed in [28, 29] require a complicated modification for the inherited sets especially when adding a get2 method. The Adaptive-Arena provides an elegant simple solution for the addition of new methods like get2 and gget that have been presented in [28, 29]. Languages such as POOL-T [3] and ABCL/1[28] support only a single thread of execution within the concurrent object; concurrent invocations of methods are always serialized.

In DRAGOON[6], synchronization code is defined in a separate class called the behavioral class. One of the limitations for DRAGOON[6] approach is that the behavioral classes can not be extended. Also, the default evaluation of synchronization counters in DRAGOON[6] may have a negative impact on performance. Similar semantics to the synchronization counters can be achieved in the Adaptive-Arena through the use of the scheduling expressions that will be evaluated when needed only.

5. Conclusion

In this paper, we presented a hierarchical approach for concurrent object-oriented programming. Our approach has been the first to address the issue of the separation between synchronization and scheduling abstractions and the functionality control of the concurrent objects. These components are specified as first class entities in our model. Our specification mechanism for these entities minimizes changes that have to be made in the superclasses. Changes in the behavior of a concurrent object are localized to the synchronization and scheduling abstractions rather than the method bodies of the sequential code for the superclass. We explained how synchronization constraints and scheduling protocols can be specified, reused, or extended just as data and operations have been extended and reused in the sequential object-oriented programming models. We also identified the root cause for the inheritance anomaly that may arise at the synchronization and scheduling levels. We showed that lack of expressive synchronization and scheduling constructs was the major bottle-neck for the employment of the object-oriented paradigm in the development of soft-real-time applications. The synchronization and scheduling abstractions in our model can be used to engineer adaptability in the development of the reactive/adaptive systems.

The Adaptive-Arena compiler comprising the full set of the language constructs presented in this paper is under development. Our future work involves an enhancement of the Adaptive-Arena concurrent object-oriented programming model which will facilitate the design and the development of the distributed applications .

References

[1] Agha, G. ACTORS: A Model of Concurrent Computation in Distributed Systems ,MIT Press, Cambridge, Massachusetts, 1986

[2] Aksit, M., Wakita K., Bosch, J., Bergmans, L., and Yonezawa, A., "Abstracting Object Interactions Using Composition Filters," GNR94, pp. 152-184.

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

[4] Andler, S. "Predicate Path Expressions," Proceedings of the 6th ACM Symposium on Principles of Programming Languages, Jan. 1979, pp. 226-237.

[5] Andrews, G.R. and Olsson, R. A. The SR Programming Language, Concurrency in Practice, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1993.

[6] Atkinson, C. Object-Oriented Reuse, Concurrency and Distribution: An Ada Based Approach, Addison-Wesley, 1991.

[7] Batory, D. and O'Malley, S. "The Design and Implementation of Hierarchical Software Systems Using Reusable Components," ACM Transactions on Software Engineering and Methodology, Vol. 1, pp. 355-398, Oct. 1992.

[8] Bloom, T. "Evaluating Synchronization Mechanisms," In Seventh International ACM Symposium on Operating System Principles, pages 24-37, 1979.

[9] Booch, G. Object-Oriented Analysis and Design with Applications, Second Ed., Benjamin-Cummings, 1994.

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

[11] Campbell, R. H. and Habermann, A. N. "The Specification of Process Synchronization by Path Expressions," In Lecture Notes in Computer Science, No. 16, pp. 89-102. Springer Verlag. 1973.

[12] Caromel, D., "Toward a Method of Object-Oriented Concurrent Programming," Communications of ACM, Vol. 36, No. 9, Se. 1993, pp. 90-102.

[13] DoD 83. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, Washington, DC, 1983.

[14] DoD 95, Ada 95 Reference Manual. ANSI/ISO/IEC - 8652: 1995, U.S. Government, 1995.

[15] Elrad, T. "Comprehensive Race Controls: A Versatile Scheduling Mechanism for Real-Time Applications," In Proceedings of the Ada Europe Conference, ed. Angel Alvarez, June 13-16, 1989, Madrid, Spain, Ada The Design Choice, Cambridge University Press, UK, June 1989.

[16] Elrad, T. and Verun, U. "Reflective Synchronization and Scheduling Control in Concurrent Object-Oriented Programming Languages," In Proceedings of the OOPSLA'93 Reflection Workshop, Washington, DC, 1993.

[17] Fayad, M. and Cline, M. "Aspects of Software Adaptability," Communications of the ACM, Vol. 39, No 10, October 1996, pp. 58-59.

[18] Francez, N. and Forman, R. Interacting Processes, Addison-Wesley, 1996.

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

[20] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

[21] Gehani, N.H. and Roome, W.D. The Concurrent C Programming Language, Silicon Press, Summit, NJ, 1989.

[22] Gehani, N.H. "Capsules: A Shared Memory Access Mechanism for Concurrent C/C++," IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 7, 1993, pp. 795-812.

[23] Geib, J. and Courtrai, L.. "Abstractions for Synchronization to Inherit Synchronization Constraints," Presented at the ECOOP'92 workshop on object-based Concurrency and Reuse, June 1992.

[24] Headington, M. and Oldehoeft, A. "Open Predicate Path Expressions and Their Implementation in Highly Parallel Computing Environments," Proceedings of the International Conference on Parallel Processing, pp. 239-246, 1985.

[25] Hoare, C.A.R. "Monitors: An Operating System Structuring Concept," Communications of the ACM, Vol. 17, No. 10, October 1974, pp. 549-557.

[26] Johnson, R. Foote, B. "Designing Reusable Classes," Journal of Object-Oriented Programming, Vol. 1, pp. 22-35, June.July 1988.

[27] Lohr, K.-P. "Concurrency Annotations Improve Reusability," Proceedings of the Conference on the Technology of Object-Oriented Languages and Systems , Oct. 1992, pp. 53-62.

[28] Matsuoka, S. and Yonezawa, A. "Analysis of Inheritance Anomaly in Object- Oriented Concurrent Programming Languages," In Research Directions in Concurrent Object-oriented Programming, eds. G Agha, P. Wegner, and A. Yonezawa, MIT Press Cambridge, MA, 1993, pp. 107-150.

[29] Matsuoka, S.,Taura, K., and Yonezawa, A. " Highly Efficient and Encapsulation Re-use of Synchronization Code in Concurrent Object-Oriented Languages," In Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93), pp. 109-126, Washington, DC, October 1993.

[30] Meyer, B. Object-Oriented Software Construction. Prentice Hall International Series in Computer Science. Prentice-Hall, 1988.

[31] Olderog, E. and Apt, K. "Fairness in Parallel Programs," ACM Transactions on Programming Languages and Systems, Vol.10, No. 3, pp. 420-455, July 1988.

[32] Schmidt, D., "The Adaptive Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications," In Proceedings of the 12th Annual Sun Users Group Conference, San Francisco, CA, pp. 214-225, SUG, June 1994.

[33] Shaw, M. "Abstraction Techniques in Modern Programming Languages," IEEE Software Vol. 1, No. 4, pp. 10-14. October 1984.

[34] Stroustrup, B. The C++ Programming Language. Second Edition. Reading, MA: Addison-Wesley, 1991.