Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications [*]

Cristina Videira Lopes [**] , Karl J. Lieberherr

College of Computer Science
Northeastern University
Cullinane Hall
Boston, MA 02115, USA
email: {crista, lieber}@ccs.neu.edu

Abstract. This paper presents a programming model for concurrent object-oriented applications by which concurrency issues are abstracted and separated from the code. The main goal of the model is to minimize dependency between application specific functionality and concurrency control. Doing so, software reuse can be effective and concurrent programs are more flexible, meaning that changes in the implementation of the operations don't necessarily imply changes in the synchronization scheme (and vice-versa). We make an analysis of concurrent computation, review existing systems and their inherent limitations, and discuss the fundamental problems in abstracting concurrency. Then we propose a solution based on lessons learned with adaptive software, introducing the concept of synchronization patterns. The result is a programming model by which data, operations and concurrency control are minimally interdependent.

Keywords: Concurrency, synchronization, object-orientation, reusability, adaptiveness.

1 Introduction

This paper describes a new approach to concurrent object-oriented programming by which synchronization schemes are expressed by a mechanism external to the programming language itself. In fact, we separate the basic behavior of the applications from their concurrent issues, introducing a new level of abstraction in object-oriented programming. We call this new level the **adaptive** level. When programming adaptive applications, behavior is described independent of any concurrent activities, and concurrency control is described with minimal assumptions on the operations; then we generate a complete and correct object-oriented program from the adaptive constructs. Doing so, the basic semantics of

one application remains unchanged for a family of different implementations that can run in sequential or concurrent form. Also, several synchronization schemes can be tested without modifying the implementation of the operations. At the same time, one particular synchronization scheme may be reused by several applications. Additionally, compatibility with the existing programming environment is maintained, because the result of this method is an ordinary object-oriented program. An overview of our approach can be seen in Fig. 1. The generation of object-oriented programs from high level descriptions of their parts can be automated.
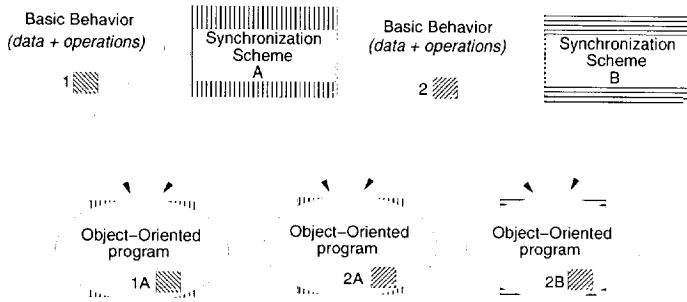

Fig. 1. Separating concurrency schemes from programs.

In this paper we want to address two questions. One first question is whether concurrency can be abstracted from applications at all. Arguments may be raised stating that some applications are inherently concurrent, and it will be impossible to separate their basic behavior from the fact that their execution includes several competing components. We think that in all cases concurrency can and should, in fact, be abstracted. The second question is how this abstraction can be implemented; that is, to find a convenient high-level language, which must be able to describe the synchronization scheme and the application algorithms at the same time, without imposing a totally new programming model at the object-oriented level.

1.1 Concurrency Revisited

Concurrency includes two opposite forces: collaboration and competition. On the one hand, concurrent applications allow the existence of a set of collaborating processes[3], and this collaboration represents performance gains, high availability, group work, etc. On the other hand, collaborating processes will eventually share resources, and therefore conflict; as a consequence, synchronization mechanisms

---

[3] Many words have been used in the literature: heavy-weight or light-weight process, task, thread, etc. Our concept of *process* includes all of those terms.

must be provided. In many cases, defining the collaboration and the synchronization schemes may be the most complex part of a concurrent application.

The issue related to collaboration is how to describe the initial and dynamic configuration of applications. Usually the operating system (or some process library) provides primitives for process creation; the way programmers have been dealing with this problem is to include a call to *createProcess* at some points in the code - the initial configuration probably hard-wired in the first lines of code. Methodologies for configuration have been proposed (for example, CSP [9]), but they usually deal with the problem at a very low-level of abstraction. The challenge is to find high level mechanisms which express initial and dynamic configuration independent from the application itself.

Concerning competition, any application must deal with situations in which two or more processes are accessing the same data at the same time. Again, the operating system (or some library) usually provides primitives for mutual-exclusion, but the direct use of those primitives tends to increase the complexity of programs. The challenge here is to find high level constructs which express the additional complexity introduced by competition, i.e. expressing process synchronization with minimal dependencies over the other aspects of the applications.

From the process's point of view we can identify concurrency from four different perspectives:

- *Process-to-processor*: what processes should execute in what processors.
- *Process-to-process*: how and when processes create other processes.
- *Process-to-data*: what data should be available and under what conditions.
- *Process-to-function*: what functions do processes execute and under what conditions.

The systematic handling of these relations is a challenge, in part due to the fact that their domains are not clearly disjoint. In order to simplify concurrent applications, it would be extremely useful to have a mechanism for expressing these relations independent of (or separated from) the code of the application. This is precisely what we propose to do: abstracting concurrency from the basic behavior of applications, so that applications are easier to program, understand and modify. In this paper, we abstract the *process-to-function* relation, due to its relevance to the problem of process synchronization.

1.2   Object-Oriented Programming Revisited

The benefits of object-oriented programming are widely known and need not be repeated. What's not so exposed is how object-oriented programs can be different from procedural programs in a very nasty way. In the object world, methods are not just what functions used to be in the procedural world. Several studies (for example, [24, 23]) have shown that methods tend to be very small, most of them serving as simple bridges to other methods. This is a natural consequence of encapsulation, and is encouraged by style guidelines for good programming [12]. Another problem is that all these little methods are explicitly attached to

classes, introducing an implicit commitment to maintain each method's code dependencies on its own class and on the classes which are referred in the code. These characteristics have two undesirable consequences: (1) understanding each class's functionality is easy, but understanding programs as a whole can be very hard; and (2) with relations between classes changing frequently, the effort to maintain the code can be substantial.

Due to the proliferation of small methods in object-orientation, models that were perfectly appropriate for parallel and distributed procedural computing are not necessarily good for parallel and distributed object-oriented computing. The complexity of the solutions proposed so far for concurrent object-oriented programming is a consequence of this disadjustment.

1.3   Adaptive Programming

Adaptive programming [11, 13, 15, 14, 21] tries to solve the problems of object-oriented programming by describing programs in a level *above* object-orientation. Adaptive programs are defined by two **building blocks**: the *structural block*, implemented by *class dictionary graphs*, which describes relations between classes; and the *behavioral block*, implemented by *propagation patterns*, which describes the operations. The building blocks are only loosely coupled with each other. This is what makes applications *adaptive*: changes in one block don't necessarily imply changes in the other. Specialized tools produce object-oriented programs from those building blocks. Adaptive applications are programmed from a global perspective, and not from each class's role in the application. In fact, the complete definition of a class - the set of methods - is not made at programming time, but it is delayed until what is called *propagation time*, where the appropriate methods are assigned to the appropriate classes.

In the adaptive method, *reuse* is not restricted to *inheritance*. Although the adaptive method makes full use of inheritance (just as it makes full use of parametrization, and all the other primitive notions of "reuse"), it allows the reuse of one building block for many different implementations of the other. In other words, the same propagation pattern can be reused for different class dictionary graphs (and vice-versa).

Our goal of abstracting concurrency from object-oriented programs can be achieved by extensions to the adaptive method. Adaptive concurrent applications are defined by the two previous building blocks and a new one, the *concurrency block* which describes the synchronization scheme between the several processes. Synchronization schemes are implemented by **synchronization patterns**. Object-oriented programs are automatically generated from these three building blocks.

The remainder of this paper is organized as follows. In section 2 we survey the most recent proposals in concurrent object-oriented programming. Section 3 discusses the conceptual problems of abstracting the process-to-function relation, revisiting well-known examples of concurrent situations. Then, in section 4 we present a possible implementation of our proposal, integrated in the generic line of adaptive software. Finally, section 5 states the conclusions and future work.

2 Directions in Concurrent Object-Oriented Programming

Researchers have tried to merge object-oriented programming and concurrency for many years, but so far there is no commonly accepted mechanism for concurrent object-oriented programming. Although the similarities between *objects* and *processes* seem obvious, in part due to the notion of *encapsulation*, the union between the two brings about many questions for which there exist no definitive answers.

A recent discussion of the state of the art can be found in Agha et al. [2]. Table 1 summarizes the main characteristics of some of the most significant solutions proposed so far. Those characteristics are: (1) atomicity *vs* non-atomicity of objects; (2) active objects *vs* uniform object model; and (3) implementation strategy: class libraries *vs* language extensions *vs* new languages.

	atomicity	active objects	implementation strategy
ABCL/1 [25]	Required for *serialized* objects	Yes	New language
CEiffel [17]	Not required	No	Extensions
Eiffel [20]	Yes	No	Extensions
Eiffel// [6]	Yes	Yes	Extensions+library
Emerald [5]	No	No	New language
Maude [19]	Yes	No	New Language - Integration with *rewriting logic*
POOL/T [3]	Yes	Yes	New language
Sina [4]	Not required	No	New language - Integration with *composition filters*

Table 1. Characteristics of concurrent object-oriented languages.

Researchers who propose atomicity of objects claim that allowing only one process at a time to execute operations on an object is a *natural* integration of objects and concurrency [3, 6], and that it is required for provability purposes [20]. Those who propose a more permissive solution claim atomicity is not always required nor desired [17].

Languages that distinguish between active and passive objects associate a special method (the *body*) to active objects. The *body* is usually a loop serving requests under certain conditions. *Passive* objects, on the other hand, don't have the *body* and can only be accessed by one active object. This approach is a natural extension of the client/server model for object-based languages, although lately it has been integrated with languages supporting inheritance. When applied to typical object-oriented applications, this approach may lead to severe performance loss, because it forces many small objects to be processes. Moreover, the distinction between active and passive objects breaks the uniformity

that exists in sequential object-orientation. In the uniform object model, activity is trivial; that is, there are no special recipients of messages, all objects can be accessed in the same way, simply by invoking one of their methods.

In relation to the implementation strategy, the solutions vary from the use of class libraries to entirely new programming languages. Class libraries are very attractive, since they do not modify the existing programming language. They are also very flexible, because classes in those libraries can provide a clean interface to low-level mechanisms (tasks, locks, semaphores, timers, etc). However, all the synchronization issues have to be solved explicitly by the programmer. This increases the complexity of programs, and lacks to provide any kind of systematic approach to the concurrency issues of applications. Although not shown in table 1, some examples of handling concurrency with class libraries are COMANDOS [22] and the Eiffel-based work in [10].

The second implementation strategy takes a sequential language and defines some extensions to it, either by adding new keywords (as in Eiffel [20]), or by inheritance from special classes (as in Eiffel// [6]), or even by special comments in the code (as in CEiffel [17]). In any case, the compiler is modified, but the programming environment is basically the same. However, it is clear that the more extensions are defined, the more flexible programs become[4].

The third implementation strategy defines new programming environments (programming languages and support systems) explicitly designed for solving the problems of concurrency. Some examples are shown in table 1. Another example of this approach is the work proposed by Agha et al. [1], based on the *Actor* model, which defines *synchronizers* - special *actors* that handle synchronization within a group of *actors*. The introduction of entirely new programming paradigms is fascinating, and can effectively solve most of the problems. However, when it implies totally new programming environments it tends not to be accepted by a large community of programmers.

One of the serious problems detected in concurrent object-oriented programming is the so-called *inheritance anomaly*: the synchronization scheme of one class, in general, cannot be effectively inherited without non-trivial class redefinitions. This phenomenon has been extensively pointed out in the literature, and compromises severely the *reuse* of classes by inheritance (Matsuoka et al. make a detailed analysis of the problem in [18]). Most proposals for concurrent object-oriented programming presented so far, suffer, to a certain degree, from the inheritance anomaly. We will come back to this issue in section 4.4.

The solution we propose does not impose atomicity of objects and does not impose the existence of active objects, although the high-level language constructs presented here could be mapped into languages with such characteristics. The main contribution of our work is concerned with the implementation strategy. We wanted to provide a high level of abstraction without defining an entirely new programming paradigm at the object-oriented level. We achieve this by using **synchronization patterns**, from which object-oriented code is generated.

---

[4] Compare, for example, [20] with [17].

3 Abstracting Concurrent Situations

We next give some examples of typical concurrent situations and show how concurrency can be separated from the basic behavior of these applications. At this point, we will not use any programming language in particular, but just pseudocode describing the main lines. The goal is to identify the basic requirements in abstracting concurrency, and we will do so by identifying, for each case, the three building blocks: structural block, behavioral block and concurrency block.

3.1 The Dining Philosophers

In the classical formulation of the dining philosophers problem [7], there are N philosophers sitting at a circular table and N forks on the table, placed between the philosophers. Philosophers execute a never ending loop of thinking and eating; to eat they need both forks, the left one and the right one, which are shared with their neighbors. There is the possibility of deadlock - when they all grab the left fork at the same time and then try to grab the right fork. One widely known solution is to make the allocation of both forks as a globally indivisible operation. The application blocks for this problem are shown in Fig. 2.

Behavioral Block:	**Structural Block:**

Philosopher does:
    Loop is
       forever
          Think
          Eat
    Think is
       // Think for a while
    Eat is
       TakeForks
       // Eat for a while
       PutForks
    TakeForks is
       left.PickUp
       right.PickUp
    PutForks is
       left.PutDown
       right.PutDown
Fork does:
    PickUp is // *going up*
    PutDown is // *going down*
	Philosopher knows-of:
	       left, right : Fork
	Fork knows-of: // *something*
	
	Concurrency Block:
	
	add-structure:
	    Fork knows-of:
	       state : State
	TakeForks@Philosopher is exclusive
	    requires:
	       left.state == FREE and
	       right.state == FREE
	    false → repeat
	    effect:
	       left.state := TAKEN;
	       right.state := TAKEN;
	PutForks@Philosopher
	    effect:
	       left.state := FREE;
	       right.state := FREE;

Fig. 2. *The dinning philosophers problem.*

Whatever function philosophers are supposed to do, it will be independent from the fact that there are N of them doing the same thing. In other words, *when* they do it may depend on external factors, but *what* they do is independent of the rest. Deadlock is a consequence of configuration (in fact, if there are 2N forks on the table, there will never be any competition).

We achieve synchronization by mutually excluding processes at operation *TakeForks*, meaning that there will be only one philosopher executing this code at one time. Moreover, to execute operation *TakeForks* both forks must be free. Therefore, for synchronization purposes, we must associate a *state* with each fork, so that we can determine if some philosopher is using it or not. In fact, this state is not a basic characteristic of *Fork*, but just an additional information for synchronization purposes.

The operation *TakeForks* must be it exclusive; moreover, it can only be executed when both forks are free. When this requirement cannot be satisfied, the presented scheme decides to make an active wait. Other policies could be used, but this one, although expensive in terms of CPU cycles, is simple and expressive enough for demonstration purposes. The point is that the reaction to an unsatisfied constraint is an important part of the synchronization scheme; a default action, such as "wait until the constraints can be satisfied", may not be appropriate for all situations.

3.2   The Bounded Queue

The second widely known concurrency example is the bounded queue. In the classical formulation of this problem, clients invoke *put* and *remove* operations over Queue objects (let's assume, for the time being, they are LIFO queues). Moreover, performing a *put* on a FULL queue or a *remove* on an EMPTY queue implies blocking the client until the state of the queue changes. The solution is also widely known, and it follows a *monitor* approach [8] to each Queue object. In this case, we can define the application blocks shown in Fig. 3.

Structural Block:	Concurrency Block:
Queue knows-of:	regions:
max : Number	$C1$: per-object
list : ListOfObjects	**Remove@Queue** is exclusive $C1$
	requires:
Behavioral Block:	*list not empty*
Queue does:	false $\rightarrow$ wait
Put(Object o) is	**Put@Queue** is exclusive $C1$
// Insert one object in the list	requires:
Object Remove() is	*list not full*
// Remove and return the newest	false $\rightarrow$ wait

Fig. 3. The bounded queue problem.

In the case of sequential programming, the implementation of the operations should avoid erratic situations; that is, before inserting/removing an object to/from the queue, a test should be performed on the number of elements in the queue. In general, *functions should not rely on any concurrency scheme* to prevent errors. This rule is justified by the following: in case no concurrency exists, behavioral errors can still occur, and their detection should be independent from all the rest. The absence of those tests is an optimization, obvious in this case, but not so obvious in general.

As for the synchronization scheme, only one process at a time should be trying to modify the queue (either inserting or removing elements): *remove* and *put* are mutually exclusive. The granularity of this exclusion is by object, that is, what's happening in one queue object is independent of what's happening in all other queue objects. This means that, independent of the particular implementations of mutual exclusion, there must be mechanism of differentiating among several exclusive regions. In Fig. 3, $C1$ is the name of the exclusive region. Removing from an empty queue or inserting into a full queue suspends the caller by a blocking wait (instead of a dynamic one, as in the philosophers example).

3.3 Observers

The observers example models a situation in which the execution of operations on objects of type A is supervised by some object of type B (possibly executing in another process). The behavior of both types of objects is independent of the the fact that any operation on A objects must trigger some operation on some B object. Although this is related to synchronization, it is the opposite of mutual exclusion, because it triggers parallel executions instead of inhibiting them. This situation can be abstracted as shown in Fig. 4.

Structural Block:	Concurrency Block:

A knows-of: *//something*
B knows-of: *//something*
 	
Behavioral Block:
 	
A does:
   f1(...) is ...
   f2(...) is ...
   ...
B does:
   Observe (Object obj, String s) is
      print (obj, s)
	add-structure:
	   A knows-of: *obs : B*
	f1@A
	   on-entry:
	      obs.observe(this_object, "IN F1")
	   on-exit:
	      obs.observe(this_object, "OUT F1")
	f2@A
	   on-entry:
	      obs.observe(this_object, "IN F2")
	   on-exit:
	      obs.observe(this_object, "OUT F2")
	   ...

Fig. 4. *The observer problem.*

We can think of more sophisticated examples of observers, such as debuggers, for which we could define a similar kind of interaction. The point is that in most cases interaction between objects and processes is independent of what the objects do, and therefore it can be abstracted, as we've just done in this informal way.

3.4 Discussion

The examples seen so far allow us to identify a set of requirements related to the problem of abstracting the *process-to-function* relation. In general, in the concurrency block we need to be able to express the following:

- Additional structure: the basic application structure may need extra parts or even new classes for synchronization purposes. This is the case of *State* in the dining philosophers example.
- Additional operations: in order to achieve synchronization, it may be necessary to define new methods. For example, in the bounded queue, determining the number of elements in the list may involve a new method for class Queue.
- Mutual exclusion sets: it may be necessary to name (and therefore distinguish between) mutually exclusive regions. Exclusion is defined on an object basis. It would be extremely useful to be able to define exclusion in terms of groups of objects.
- Method identification: in all the previous examples, it is necessary to identify which methods will be synchronized. This is the inevitable dependency of the synchronization scheme over the application operations. However, it is clear that we may not need to explicitly synchronize all of those operations.
- Preconditions: it may be necessary to state a set of conditions that must evaluate to true before operations are executed. That is the case of the dining philosophers and of the bounded queue.
- Reaction to false preconditions: this is what defines the call semantics of the method. A default WAIT (until preconditions become true) semantics may not be appropriate in all cases; instead, it may be better to return immediately or to wait during a limited time. The synchronization mechanism must be expressive enough to accommodate many different reactions. In the case of the dining philosophers, we decided to make an active wait (repeat), whereas in the bounded queue we decided for a blocking wait.
- Effects: in the observers example, we've seen how the execution of some method may trigger other actions. In general, effects can be separated into *on-entry* and *on-exit* actions; *on-entry* actions will be performed as soon as the process starts executing the operation, and *on-exit* actions will be performed just before the process finishes the execution of the operation.

4 Towards a Solution

Next we make a more formal approach to the programming model exposed in the previous section. The solution we propose is integrated in the generic

line of adaptive programming described in section 1.3. Adaptive programs are parametrized by a structural block (implemented by class dictionary graphs) and a set of functions (implemented by propagation patterns) which involve groups of classes without explicitly referring to all of them. We can view the role of the adaptive tools as a function:

$$F : propagation\,patterns \times class\,dictionary \longrightarrow OO\,program$$

Moving into concurrent computation, we extend this function by the following:

$$F' : synchronization\,patterns \times propagation\,patterns \times class\,dictionary \longrightarrow OO\,program$$

Synchronization patterns, propagation patterns and class dictionaries are only minimally interdependent. The word *minimally* means that the connections between these building blocks include only the information that is absolutely necessary, and ignore all that is not important for the interaction. Several examples of minimal dependency between class dictionaries and propagation patterns can be found in [16, 15, 14]. As for synchronization patterns, we will see how they can be reused, and how they solve the inheritance anomaly.

4.1   Assumptions and Requirements

As we mentioned earlier, we make no assumptions about the underlying programming paradigms associated with the particular object-oriented languages and systems. Defining class dictionaries, propagation patterns and synchronization patterns is independent of whatever facilities the lower levels provide. The composition of these three building blocks will be mapped accordingly. Nevertheless, for some languages/systems the mapping will eventually be more restrictive than for others. Moreover, the more restrictions that exist on the object-oriented language, the less flexible the applications can be. For example, the expressiveness that we may have with synchronization patterns may be lost when generating a program in Eiffel//, since it imposes the model of active objects. In fact, the ideal object-oriented level will make no restrictions concerning the object/process space: object and process dimensions should be orthogonal. However, at the object-oriented level some minimal requirements must be observed:

- Object identifiers must be available; in particular, the *current object* identifier must be held in some variable. This is accomplished by all widely used object-oriented languages, and needs no further justification.
- Process identifiers must be available; in particular, the *current* process identifier must be held in some variable. This is not so obvious, but it is justified by the fact that the low-level synchronization code may need to identify the processes (for example, putting them on a waiting list and later removing them from that list).
- The system/language must provide some mechanism for mutual exclusion.
- The system/language must provide some mechanism for blocking/unblocking processes.

Note that the above requirements are very permissive. They can be satisfied by any object-oriented language (say, C++) linked with a thread library and

with a very simple *Lock* class. Since we make no assumptions on the object-process relation we need two different identifiers. The unit of computation, that is, the object and the thread of control, is given by the pair *(objId, procId)*.

4.2 Synchronization Patterns

The concurrency block identified in section 3.2 is expressed by a high-level construct called *synchronization pattern*. Figure 5 shows the synchronization pattern for the bounded queue problem. It follows the informal description given in section 3.2, and uses the notation seen in previous papers [16, 15]. Keywords are in capital letters; structure is defined in terms of additional parts and/or classes, which can have a symbolic name inside the synchronization pattern; primitive programming language code is placed between (@ and @) (in this case we are using C++).

```
SYNC_PATTERN sync_A
  ADD_STRUCTURE        // additional structure
      // empty
  ADD_FUNC             // additional operations
    OPERATION int get_n_elements()
      TRAVERSE FROM Queue TO ElementList
        WRAPPER ElementList
          SUFFIX (@ return_value = this->num_of_elements(); @)
  MUTEX                // mutual exclusion names
    PER_OBJECT x1
  SYNC                 // synchronization scheme
    OPERATION Element *Remove()
      AT Queue EXCLUSIVE x1
        REQUIRES
          (@ this->get_n_elements() != 0 @)
          FALSE (WAIT)
        ON_EXIT ((@ cout << ''Leaving Remove''; @))
    OPERATION void Put (Element *e)
      AT Queue EXCLUSIVE x1
        REQUIRES
          (@ this->get_n_elements() < max @)
          FALSE (WAIT)
        ON_ENTRY ((@ cout << ''Entering Put''; @))
```

Fig. 5. Synchronization Pattern for the bounded queue problem.

In general, synchronization patterns contain 4 blocks: 1) definitions for additional structure; 2) definitions for additional operations; 3) declaration of mutual exclusion names; 4) definition of the synchronization scheme. The first three blocks correspond to the first three requirements in section 3.4. The fourth block

defines the synchronization scheme in terms of mutual exclusion between operations, preconditions, reaction to false preconditions and effects. In the case of the bounded queue, we define an exclusion named $x1$: for the same object, we want to avoid the simultaneous execution of *Put* and *Remove*.

4.3   Generating an Object-Oriented Program

We now give an illustrative example on how the three building blocks of the bounded queue application can be mapped into a C++ program. Given the synchronization pattern in Fig. 5 and the class dictionary and propagation patterns in Fig. 6, the resulting object-oriented code for class Queue is shown in Fig. 7 (only the method *Put* is shown; *Remove* is very similar).
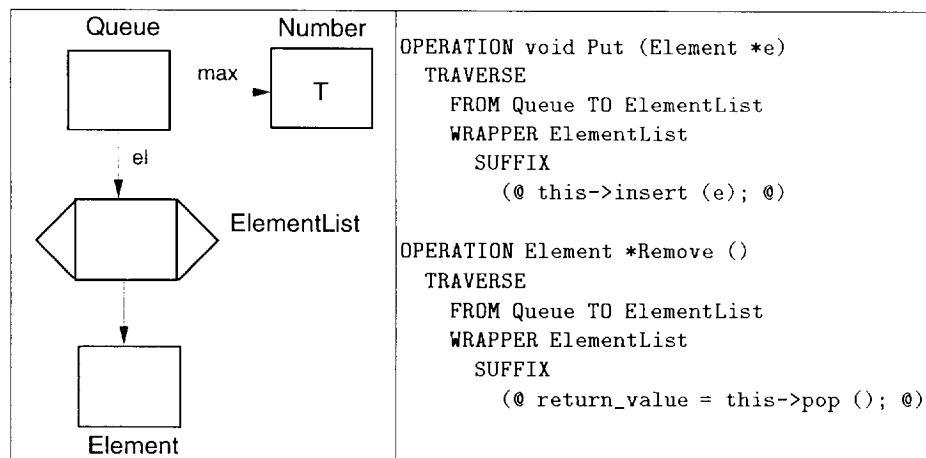
Fig. 6. Class dictionary graph and propagation patterns for the bounded queue application.

Figure 7 is self-explanatory. In this particular mapping into object-oriented code, we assume that locking is possible through invocations to a special object *LockManager* which blocks the requesting process until the required lock is available, but any other mechanism could be used. We also associate with each Queue object a list of processes which are waiting to execute over the object. Note that this mapping to C++ is just one of many possible mappings. However, the particular mapping that we chose can always be automated; that is, each part of a synchronization pattern will generate code according to a specific algorithm. In this example, the code generation algorithm for any operation referenced in a synchronization pattern observes the following guidelines:

- If the operation is defined as EXCLUSIVE, a local variable containing the lock name is defined. The name contains at least the synchronization pattern

```
class Queue {                    void Queue::Put (Element *e) {
    ActivityList *_al;               char lockname[50];                    lock naming
    ElementList *el;                 sprintf (lockname, "sync_A_x1%x", this);   (PER_OBJECT x1, within sync_A)
    DemNumber *max;
public:                              while (1) {                            for when preconditions evaluate to false
    void Put (Element *);
    Element *Remove ( );              LockManager->Lock (this_activity, lockname);   EXCLUSIVE x1
};
                                      if (this->get_n_elements() < max) {   REQUIRES    (preconditions = true)
                                        cout << "Entering Put";             ON_ENTRY

                                        el->Put (e);                        default code (only this line!)
                                                                            ON_EXIT

                                        if (_al->n_elements( ) != 0)
                                          _al->pop( )->Resume( );           unblock some activity
                                        LockManager->Unlock (this_activity, lockname);  over this object
                                        return;
                                      }
                                      else {                                FALSE    (preconditions = false)
                                        _al->insert (this_activity);
                                        LockManager->Unlock (this_activity, lockname);  WAIT  (forever)
                                        this_activity->Suspend ( );
                                      }
                                    }
                                  }
```

Fig. 7. The resulting OO program.

name; if the mutual exclusive region was given a name ($x1$, for example), then that name will also be part of the lock name; if the mutual exclusion is defined per-object, then the lock name also includes the object unique identifier. If the operation is not EXCLUSIVE, no lock name variable is needed.

- For an EXCLUSIVE operation, if its reaction to a false precondition is not QUIT, a loop is generated (*while (1)* {), so that when the process is resumed (or repeats) it will try to execute the code again. If the reaction is QUIT, or if the operation is not EXCLUSIVE, the loop is not necessary.
- Mutual exclusion is achieved by invoking the *LockManager*, given the process identifier and the lock name. This invocation blocks the process until the requested lock is available.
- The REQUIRES clause generates a test. For the *true* portion of the test, the following code is generated: ON_ENTRY actions, the default code, ON_EXIT actions, code to unblock some blocked process and code to free the mutual exclusion lock (in this order). For the *false* portion of the test, the code generator produces code that implements the required call semantics (wait, wait for some time, etc).

4.4 Inheritance Anomaly

An important point is the relation between synchronization patterns and class inheritance. Three situations may occur:

1 - The synchronization scheme is defined for method M of class C and M is defined directly in class C. This is the simplest situation, and it occurred in the example of Figs. 5 and 6. The code generator defines M for C according to the algorithm explained in section 4.3.

2 - The synchronization scheme is defined for method M of class C and M is inherited from a super-class of C. This is a specialization of the synchronization scheme for the inherited operation. In this case, the code generator must redefine method M for class C.

3 - The synchronization scheme is defined for method M of class C, but a sub-class of C, C_sub, redefines M. This is a specialization of the operation for the inherited synchronization scheme. In this case the code generator must define method M for class C and also redefine method M for class C_sub.

These three situations have been identified before, in relation to the inheritance anomaly that exists in most concurrent object-oriented languages [18]. The anomaly is related to the inheritance mechanisms and so far there is no commonly accepted solution for this problem. In fact, the solutions proposed in the literature are either very complex or very restrictive or aim for the minimization of the anomaly rather than its elimination[5]. We think that solving inheritance anomaly at the object-oriented level is not only very hard, but it may not be the proper thing to do. For objects, the synchronization scheme is embedded in their behavior; if a sub-class C_sub defines more operations than its super-class C, then it's natural that the overall behavior of instances of C_sub may be affected by that addition. The anomaly is not related to the objects themselves, but it's a characteristic of the software model - reuse by inheritance. The *weakness* associated with the anomaly appears when we try to *identify* "software reuse" with "inheritance". However, for a different concept of reuse, the inheritance anomaly may not be considered a *weakness*, but simply a characteristic of object-oriented languages when they try to model synchronization.

Our solution does not attempt to solve the anomaly at the object-oriented level; on the contrary, methods may be redefined for sub-classes solely because of synchronization needs. However, all of that is made by a code generator, and therefore, is transparent to the programmer. Reuse is achieved at a higher level of abstraction, as we will see next.

4.5   Reuse of Synchronization Patterns

Since synchronization patterns are based on minimal knowledge about other building blocks of the application, their reuse can be effective. Suppose that we want to include more functionality in bounded queue objects by defining two new operations *first* and *last* to class Queue (or some sub-class of Queue). These operations should either return the first and the last elements of the queue (respectively) without removing them or return NULL if the queue is empty. Note that these are just query operations that don't change the state of the object. Adding this additional functionality is as simple as defining two new propagation patterns, and nothing else needs to be modified: queries can execute concurrently with any other operation on the queue. Also, suppose that we want to modify the structure and implementation of Queue objects by allowing them

---

[5] The solution proposed in [18], for example, achieves only minimization.

to be either FIFO or LIFO. This change implies modifications both to the class dictionary and to propagation patterns, as it is shown in Fig. 8. For subclass Fifo, *Remove* must be redefined. However, the synchronization pattern is exactly the same as the one in Fig. 5.
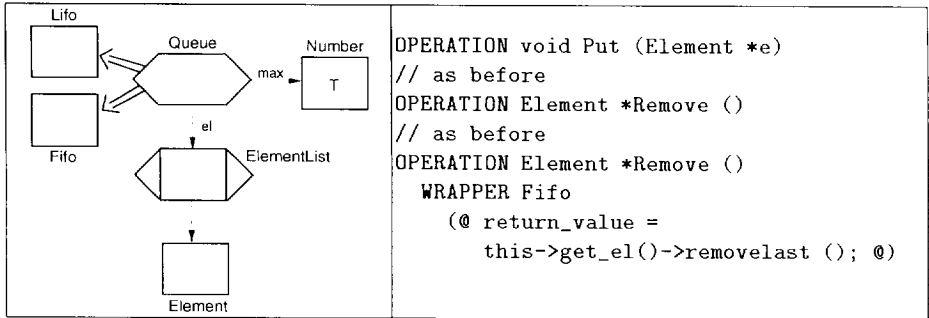


```
OPERATION void Put (Element *e)
// as before
OPERATION Element *Remove ()
// as before
OPERATION Element *Remove ()
    WRAPPER Fifo
        (@ return_value =
            this->get_el()->removelast (); @)
```

Fig. 8. New structure and implementation for Queue objects; the hexagon represents an abstract class.

Inheritance anomaly does not exist at the adaptive level. In fact, defining sub-classes with new sets of methods may involve modifications in the synchronization scheme, but it will not require the redefinition of inherited operations. At the object-oriented level, those redefinitions may occur, but they will be automatically generated. In the sequence of the example in Fig. 8, suppose that we want to define another sub-class of *Queue*, *QQueue*, with a new method *RRemove*; *RRemove* is similar to *Remove*, but it cannot be executed immediately after the invocation of *Put*. This example makes *QQueue history-sensitive* and is usually considered a difficult situation to solve concerning the inheritance anomaly. The three building blocks of the application can be seen in Fig. 9. Although a new synchronization pattern is necessary to accommodate the behavior of *QQueue* objects, the inherited methods *Remove* and *Put* don't need to be redefined for *QQueue*.

The synchronization pattern in Fig. 9 introduces the notion of inheritance applied to synchronization patterns. In this case, synchronization pattern *sync_AA* inherits from synchronization pattern *sync_A* (defined in Fig. 5). The effects of inheritance here are similar to what's usual for classes: a "sub sync-pattern" contains all the additional structure and operations, mutex's and synchronization scheme of its "super sync-pattern"; however, it may include more information and/or redefine the inherited information. The inheritance relation between *sync_AA* and *sync_A* is a natural consequence of the situation we are trying to model: the synchronization of *QQueue* objects is a *specialization* of the synchronization of *Queue* objects.
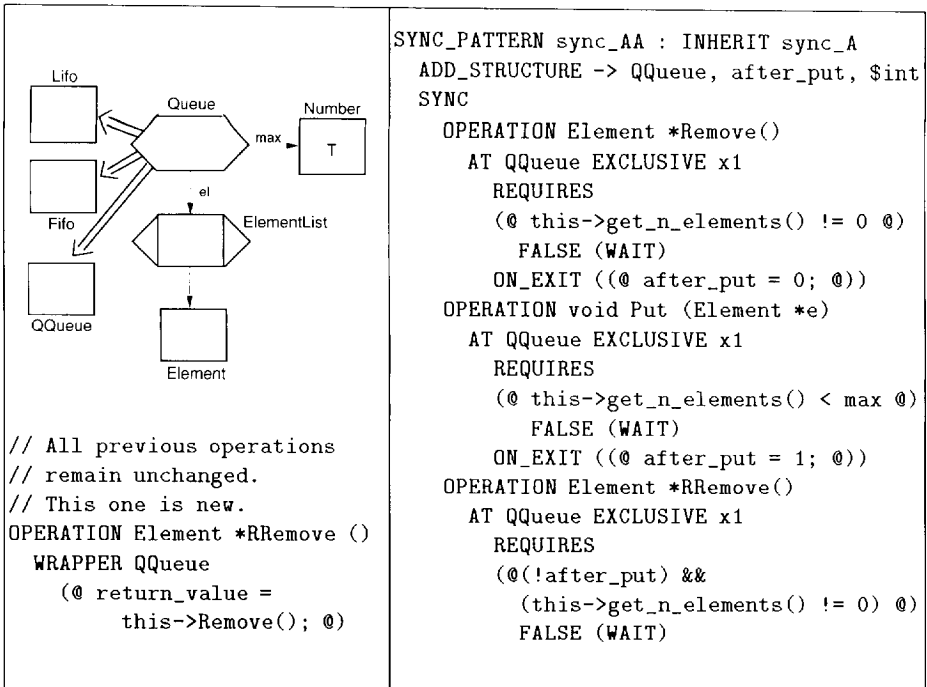
```
                                    SYNC_PATTERN sync_AA : INHERIT sync_A
                                      ADD_STRUCTURE -> QQueue, after_put, $int
  Lifo                               SYNC
  ┌───┐     Queue      Number          OPERATION Element *Remove()
  │   │      ⬡      max  ┌───┐           AT QQueue EXCLUSIVE x1
  └───┘              ▶   │ T │             REQUIRES
  ┌───┐         el       └───┘             (@ this->get_n_elements() != 0 @)
  │   │         ▼                            FALSE (WAIT)
  └───┘      ⬡ ElementList                 ON_EXIT ((@ after_put = 0; @))
  Fifo                                  OPERATION void Put (Element *e)
  ┌───┐         │                         AT QQueue EXCLUSIVE x1
  │   │         ▼                           REQUIRES
  └───┘      ┌───┐                          (@ this->get_n_elements() < max @)
  QQueue     │   │                            FALSE (WAIT)
             └───┘                          ON_EXIT ((@ after_put = 1; @))
           Element                      OPERATION Element *RRemove()
                                          AT QQueue EXCLUSIVE x1
  // All previous operations                REQUIRES
  // remain unchanged.                       (@(!after_put) &&
  // This one is new.                          (this->get_n_elements() != 0) @)
  OPERATION Element *RRemove ()              FALSE (WAIT)
    WRAPPER QQueue
      (@ return_value =
           this->Remove(); @)
```

Fig. 9. New structure, propagation pattern and synchronization pattern to accommodate QQueue objects.

5 Conclusions and Future Work

The merge of object-oriented programming with concurrency has not been smooth. The modularity and simplicity of the object-oriented paradigm for sequential computing breaks when it comes to parallel and/or distributed computing. Many attempts have been made in order to find a good solution: use of class libraries (e.g., task libraries), extensions to the sequential programming language (e.g., proposal by Meyer [20]) and even entirely new programming models (e.g., Maude [19]) are examples of how researchers have dealt with the problem.

We have presented a new approach to the problem that stands *above* object-orientation. Abstraction is achieved by using high-level language constructs which describe programs in terms of their building blocks, namely, class dictionaries, propagation patterns and synchronization patterns, but instead of promoting these constructions into special system entities, we just use them to generate the appropriate program using some existing object-oriented language. Doing so, we gain three points: (1) abstraction and reuse of synchronization schemes at the adaptive level; (2) flexibility of using *low-level* specialized classes at the object-oriented level (locks, timers, etc); (3) compatibility with the existing programming environment at the object-oriented level (compilers, debuggers, etc).

We are currently studying the enhancement of synchronization patterns,

namely the possibility of incremental redefinition of sub synchronization patterns, which will allow a more effective reuse of the synchronization scheme defined for super-classes. We are also studying how to extend the definition of mutually exclusive methods to include methods of arbitrary sets of objects.

This work is being integrated in the Demeter System^{TM}, which is under development at Northeastern University. The current version of Demeter/C++ supports only class dictionaries (edited with graphical interfaces) and propagation patterns (also animated with graphical interfaces). We are extending Demeter to allow the development of parallel and distributed applications, by defining the necessary programming abstractions, such as synchronization patterns.

Acknowledgements:

References

1. G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 1, pages 3–21. MIT press, 1993.

2. Gul Agha, Peter Wegner, and Akinori Yonezawa. *Research directions in Object-oriented programming*. MIT Press, 1993.

3. Pierre America. POOL-T: A parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–86. MIT Press, 1987.

4. Lodewijk Bergmans, Mehmet Aksit, Ken Wakita, and Akinori Yonezawa. An object-oriented model for extensible concurrent systems: the composition-filters approach. *Memoranda Informatica*, 92(87), December 1992.

5. A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. In *OOPSLA '86 Proceedings*, pages 78–86, Portland, Oregon, September 1986.

6. Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–101, September 1993.

7. Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

8. C.A.R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

9. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

10. M. Karaorman and J. Bruno. Introducing concurrency to a sequential language. *Communications of the ACM*, 36(9):103–116, September 1993.

11. Karl Lieberherr. *The Art of Growing Adaptive Object-Oriented Software.* PWS Publishing Company, 1995. To be published.
12. Karl Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, September 1989.
13. Karl J. Lieberherr, Walter Hürsch, Ignacio Silva-Lepe, and Cun Xiao. Experience with a graph-based propagation pattern programming tool. In Gene Forte et al., editor, *International Workshop on CASE*, pages 114–119. Montréal, Canada, 1992. IEEE Computer Society.
14. Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, May 1994. Accepted for publication.
15. Karl J. Lieberherr and Cun Xiao. Minimizing dependency on class structures with adaptive programs. In S. Nishio and A. Yonezawa, editors, *International Symposium on Object Technologies for Advanced Software*, pages 424–441, Kanazawa, Japan, November 1993. JSSST, Springer Verlag.
16. Karl J. Lieberherr and Cun Xiao. Object-Oriented Software Evolution. *IEEE Transactions on Software Engineering*, 19(4):313–343, April 1993.
17. Klaus-Peter Löhr. Concurrency annotations for reusable software. *Communications of the ACM*, 36(9):90–101, September 1993.
18. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 1, pages 107–150. MIT press, 1993.
19. José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In Oscar Nierstrasz, editor, *Lecture Notes in Computer Science*, pages 220–246. ECOOP'93, Springer-Verlag, July 1993.
20. Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, September 1993.
21. Ignacio Silva-Lepe, Walter Hürsch, and Greg Sullivan. A Demeter/C++ Report. *C++ Report, SIGS Publication*, February 1994. To be published.
22. Pedro Sousa, Paulo Ferreira, José Monge, André Zuquete, Manuel Sequeira, Paulo Guedes, and José Alves Marques. IK Implementation Report. Technical Report INESC-0014, ESPRIT COMANDOS Project, October 1990.
23. Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, pages 1038–1044, December 1992.
24. Norman Wilde, Paul Matthews, and Ross Huitt. Maintaining object-oriented software. *IEEE Software*, pages 75–80, January 1993.
25. A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–86. MIT Press, 1987.