Composing Multiple-Client-Multiple-Server Synchronizations

Mehmet Aksit and Lodewijk Bergmans
TRESE project, CTIT and Department of Computer Science,
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.
email: {aksit, bergmans}@cs.utwente.nl
http://wwwtrese.cs.utwente.nl

Abstract

A considerable number of language mechanisms have been proposed during the last several years, to specify and implement concurrent object-oriented programs. The major concern of these proposals is to design an expressive language that provides extensible concurrent processing and synchronization features. Almost all these efforts, however, have focused on the multiple-client-single-server model where each server determines its synchronization semantics without cooperating with other objects. We believe that object-oriented concurrent languages must not only support a single-server model, but cooperatively synchronizing servers as well. We refer to this as multi-server synchronization. This paper first classifies multi-server synchronization in five categories. The intention here is to define a framework for evaluating current approaches and identifying the requirements for designing new languages. In addition, this paper presents a composable multi-server synchronization technique, adopting the concept of composition-filters.

1. Introduction

It has been claimed that object-oriented language constructs are very suitable for expressing concurrency and synchronization since more than one object can be active concurrently [36]. In addition, inheritance can help in reusing well-designed and verified concurrent programs. This may reduce the efforts in writing concurrent applications, as concurrent systems are harder to design and implement. During the last several years, a considerable number of object-oriented languages have been introduced as a tool to develop concurrent object-oriented programs [7, 24, 11, 16, 30, 27]. These languages aim at constructing expressive and extensible concurrent programs. Since most early concurrent object-oriented languages failed in combining their concurrency and synchronization mechanisms with the inheritance structure, a major portion of the recent research activities were devoted to solve the so-called synchronization inheritance anomalies [29].

Almost all of the research activities in concurrent object programming have been focused on the multiple-client-single-server model, where synchronization semantics are determined by the server object only. A large number of applications, however, require cooperatively synchronizing server objects.

Although there have been some efforts to model coordinated behavior among objects explicitly [22, 51], these activities were mainly targeted on functionality rather than the reuse of synchronization semantics of coordination.

In this paper we present five types of synchronizations that one may need in expressing multi-server synchronization schemes. Our intention here is to present a framework for evaluating current approaches and identify the requirements of the future object-oriented languages.

To solve multi-server synchronization problems, we apply the concept of composition-filters. Composition-filters are modular and orthogonal extensions to the conventional object model. Filters can be used, for example, to express synchronization per object basis. Filters can also express meta-level architectures. Through the orthogonal composition of meta-level architectures and synchronization filters, one may implement the multi-server synchronizations in a composable way.

The following section introduces an abstract object model which is used to identify the issues in defining multi-server synchronizations. Section 3 defines the five types of multi-server synchronizations that one may need in constructing concurrent object-oriented programs. Based on this classification, section 4 evaluates the background and related work. Section 5 applies the composition-filters concept as a possible solution to express composable multi-server synchronizations.
Finally, evaluation of the proposed approach and conclusions are presented in section 6.

2. Definitions

The concept of encapsulating a shared resource as a server module which reacts to request messages from its clients forms the basis of object-based concurrent programming [36]. In the multiple-client-single-server model, the policies for the synchronization of concurrent operations invoked on the shared server object are determined by the server object only. Most object-oriented concurrent languages provide this kind of synchronization.

A bounded buffer object accessed by several client objects is a typical example of the multiple-client-single-server model. A bounded buffer object provides the operations get and put, which remove and add elements from and to the buffer, respectively. In addition to mutual exclusion, there are two boundary situations that require synchronization. In the first case if the buffer stores no elements at all, a get operation must be blocked until new elements are added. In the second case, if the buffer is filled to its limit, then no more put operations must be accepted until additional space is available.

Object-oriented concurrent programming extends the object-based model by adding inheritance-like mechanisms [24, 8, 13, 15, 26, 31, 30, 9, 10]. Most concurrent object-oriented languages, in one way or another, fail in combining their concurrency and synchronization mechanisms with the inheritance structure. This failure is referred to as inheritance anomaly [29]. Ideally, development methods and language constructs should be able to express concurrency and synchronization while maintaining a high degree of modularity, extensibility and reusability.

By distributing the synchronization functions of the single-server object, the multiple-client-multiple-server (in short multi-server) model can be realized. To identify the related issues of multi-server synchronizations and evaluate the existing approaches, a model is presented in figure 1. For simplicity, we do not make distinction between objects and classes. The model is characterized by five basic relations. The first two relations are inheritance and delegation. As shown by the figure, the synchronization specification of the server object can be composed from other objects through inheritance and/or delegation relations. Inherited and/or delegated objects can be seen as extensions of the server object. Thirdly, an object may exchange messages with its peer objects to implement a distributed synchronization. Fourth, a synchronization specification can be associated with one or more objects. Finally, some constraints can be defined on these relations. Here we restrict ourselves to ordering constraints. Ordering of message sends is supported by the conventional object model in the implementation of object's operations. Ordering of a synchronization specification is again a synchronization specification. Therefore, we only consider here ordering of compositions.

![Diagram](https://example.com/diagram.png)

Figure 1. The reference object model for the analysis of multi-server synchronization issues.

3. Categories of synchronization compositions for multiple-servers

This section discusses the five basic relations as identified in the multi-server synchronization reference object model.

3.1. Synchronization composition through inheritance

In the first case, the synchronization of the server object is composed through the inheritance mechanism. The abstract object model as shown in figure 2(a) illustrates this approach. Inheritance of synchronization specifications has been studied extensively in the literature [29, 9, 10].

Consider class LockingBuffer that inherits from both class BoundedBuffer and class Locking. Class LockingBuffer is a bounded buffer that can be locked and unlocked. If the buffer is locked no methods are accepted.
for execution, except the method unlock that removes this restriction. This example is illustrated by figure 2(b). The states of objects are shown as circles within objects. There may be two problems experienced in the realization of LockingBuffer. Firstly, the synchronization constraints that are defined by the two classes are sometimes contradictory. For example, suppose an instance of LockingBuffer is in the locked state and contains a few elements. Class Locking dictates that no method except unlock is to be accepted. According to class BoundedBuffer, however, the methods put and get should be acceptable. These conflicting constraints must be resolved. The second problem is that the synchronization constraints defined by Locking are to be applied on the methods of a subclass. These methods are not known yet when Locking is defined. It is thus important to define a form of open-endedness in the synchronization specification.

Figure 2. (a) An abstract model of LockingBuffer. Here \( n = 2 \). (b) Composition of the synchronization of LockingBuffer from classes Locking and BoundedBuffer.

### 3.2. Synchronization composition through delegation

The delegation mechanism allows objects to share behavior at the object level [25]. Figure 3(a) shows an abstract object model of synchronization compositions using both inheritance and delegation. For example, in figure 3(b), object SharedLockingBuffer inherits from BoundedBuffer and delegates to the shared object SharedLocker of class Locking. If a lock or an unlock message is invoked on one of the SharedLockingBuffer objects, then this request will be delegated to the SharedLocker object. Since delegation is defined at the object level, the state of the locker can be shared by multiple buffers. This makes it possible to distribute the locking synchronization among multiple buffer objects.

The problems that may be experienced here are similar to the previous example. In addition, since the delegated object is global, the delegation operation of buffer objects may interfere with each other. Therefore, mechanisms must be introduced to synchronize concurrent delegations.

Figure 3. (a) The abstract model of SharedLockingBuffer. (b) Schematic representation. Here, locking synchronization is shared by multiple buffers through delegations.
3.3. Cooperating synchronizations

Consider the definition of class DistributedBuffer, as shown by figure 4. The interface of DistributedBuffer is the same as the one of BoundedBuffer. If a distributed buffer object receives a get message and the buffer is empty, then it interacts with other buffer objects to retrieve an item for its client object. The message is only queued if all the cooperating buffers are empty. Similarly, if the buffer is full and a put message is received, then the buffer object tries to find a free buffer to store the item. If all the buffer objects are full, then the put message is queued.

![Figure 4](image)

Figure 4. (a) Interaction diagram and (b) composition hierarchy for coordinating buffers.

The abstract object model of DistributedBuffer is shown in figure 5. As depicted in the figure, each object has its own synchronization specification. However, objects exchange messages to implement a distributed synchronization.

There are two important aspects in realizing coordinating synchronizations among distributed servers. Firstly, coordinating servers must be compatible. An essential requirement is that servers must be functionally compatible. If functional compatibility does not exist, cooperation is not possible. If, however, procedural compatibility does not exist, but the components are functionally compatible, cooperation can only be obtained by introducing a suitable adaptation module which translates between the communication patterns used by the different servers. The second requirement is the synchronization of distributed servers (synchronization of distributed synchronizations). The cooperation can be realized in a distributed way, or in a centralized manner by using explicit coordination objects.

![Figure 5](image)

Figure 5. An abstract object model for cooperating synchronizations.

In the example, since the cooperating buffers are functionally compatible, it should be possible to implement class DistributedBuffer based on cooperating multiple bounded buffers.

3.4. Polymorphic synchronization compositions

Polymorphic synchronization composition allows the same synchronization specification to be used in different kinds of cooperation. Consider, for example, atomic transactions. Transactions attempt to maintain system consistency by dealing with possible problems due to exceptions, system failure, and multiple concurrent activities, and in most systems guarantee the permanence of the updates made during a transaction as well. One of the important characteristics of transactions is the serialization of executions [20], which requires synchronization of objects that participate into a transaction.

Figure 6 shows cooperating locking objects. If the method lock is invoked on one of the cooperating lockers, then after locking itself, this object locks all the other lockers. Similarly, the unlock operation unlocks all the lockers. The cooperation among lockers may result in inconsistencies if the distributed locking or unlocking protocols are not atomic. For example, before all the objects are locked, some other object can refer to the status of one of the lockers or may even try to unlock it. To avoid inconsistencies, an atomic transaction object can be composed with the cooperating lockers.
Atomic transactions are not only needed for cooperation among lockers but can be useful in other applications as well. For example, cooperating buffers shown in figure 4 could also adopt a transactional mechanism. The atomic transaction object has its own synchronization properties. This results in polymorphic applications of synchronization semantics of atomic transactions in different cooperation patterns. Other examples of polymorphic synchronizations are asynchronous message sending and broadcasts.

Figure 7 shows an abstract model for polymorphic synchronizations, where the same synchronization specification is associated with multiple objects. In the atomic transaction model, serialization is enforced on message executions as a synchronization constraint.

3.5. Ordered compositions

It may be desirable sometimes to order the compositions of objects. This is shown in figure 8 through the definition of class ProtectedLockingBuffer. Class ProtectedLockingBuffer delegates to object Verifier of class Login and inherits from classes Locking and BoundedBuffer.

The class Login has two methods verify and change. The method verify accepts name and password as an argument and controls whether this pair is registered in the local access-list. If not, an error condition is raised. The method change is used to add or remove names and passwords to or from the access-list. Assume that the operations of class ProtectedLockingBuffer have to be invoked according to its composition structure; first any method of Login, then Locking and BoundedBuffer are allowed.

The conventional object-oriented model does not provide any facility to order inheritance and/or
composition hierarchies as illustrated by figure 8. One possibility is to declare each composition possibility as a separate method in the subclass, and invoke the inherited or delegated methods through super calls. In our example, this would require eight method definitions. It may be clear that the number of possible method combinations can be quite large, and it would not be feasible to declare all possible combinations separately.

4. Background and related work

This section analyzes the state-of-the-art object technology in implementing multiple-server synchronizations.

4.1. Multiple-client-single-server models

As we stated before, most research in the area of concurrent object-oriented programming has focused on single server synchronization. Here we consider mechanisms where synchronization is performed by blocking the received messages at the boundary of objects, before the corresponding method is executed.

Several types of message passing semantics have been adopted by concurrent object-oriented languages. Examples are asynchronous message passing, where an object continues execution after sending a request, without waiting for a reply (supported e.g. in ABCL [36], Eiffel// [13] and Parallel Objects [14]) and the common blocking remote procedure call, where the sender of the message blocks until it receives a reply.

In addition, several types of synchronization schemes have been proposed. Two distinct approaches are respectively explicit acceptance and activation conditions. Explicit acceptance means that during of a method execution, the state of the object is considered, and based on this it is explicitly decided what message to accept next. For example ABCL/1 [36], POOL [7] and Eiffel// [13] support this approach.

Activation conditions define for each message whether it should be accepted or blocked. In some languages the activation conditions can be manipulated directly (e.g. Actor languages [1] and Hybrid [33]). In other languages and systems the activation conditions are specified indirectly, e.g. through guards (e.g. in [16], Guide [15] and Concurrency Annotations [26]), path expressions (e.g. PROCOL [11]) or state abstractions (e.g. ACT++ [24], Rosette [34] and Synchronizing Actions [32]). The proposal in [30] combines activation conditions through (polymorphic) guards with state abstractions.

4.2. Using distributed protocols within code

If the adopted programming language does not provide any appropriate language construct to define multi-server synchronization, the programmer is forced to implement synchronization, in case such a synchronization scheme is necessary. For example, class DistributedBuffer can be defined by modifying the implementations of the methods put and get, which are inherited from class BoundedBuffer. When the buffer is full and a put message is received, the buffer object may coordinate with other objects to find out a suitable buffer object to store the item. Similarly, if the buffer is empty and a get message is received, then the get method can be extended with a piece of code to retrieve a suitable item from other buffers. This way of implementing multi-server synchronization mixes the synchronization code with the application code. As a consequence, changing the distributed synchronization scheme in the subclass or changing the behavior of the method may require the redefinition of both the synchronization specification and the behavior, although this may not be necessary.

4.3. Using language keywords

A programming language may provide a set of keywords for distributed synchronization. For example, language constructs such as begin_transaction and commit_transaction can be used to designate a transaction block. Another example is coordinated termination, where the sender object sends multiple messages and then waits until it receives e.g. a single reply, or all replies (as provided in ABCL/1 [36] and Orient 84K [23]).

One of the disadvantages of these approaches is that such language constructs have to be specified within the body of the method. This results in mixture of synchronization specification with the application code. This may cause inheritance anomalies. Secondly, by using dedicated language keywords, only a fixed set of synchronization constraints can be specified.

4.4. Using languages with explicit inter-object communication modules

Several language mechanisms have been proposed in the past for modeling coordinated behavior among objects. These language designers claimed that coordination among objects must be represented as first-class objects. For example, a language construct called scripts [18] was introduced to abstract patterns of messages into a module. A script is a parameterized program section in which processes enroll in order to
participate. The concept of enrollment is similar to the subroutine call mechanism whereby the execution of the role in a given script instance is a logical continuation of the enrolling process. A script consists of formal process parameters called roles, data parameters and a concurrent program section called the body. Processes can enroll in scripts by means of enroll in statements.

In the object-oriented modeling area, contracts [21, 22] were defined to specify the contractual obligations that a set of participants must satisfy. A contract specification includes the specification of the participating objects, the contractual obligations of all participants, the invariants to be maintained by the participants and the method which instantiates a contract.

In [5], Abstract Communication Types (ACTs) were defined to model coordinated behavior among objects as a class. ACTs can be uniformly integrated with the coordinating objects using the composition-filters concept.

Both scripts, contracts and Abstract Communication Types basically aim at modeling the functional coordinated behavior among objects. Therefore, these language mechanisms fail in expressing composable multi-server synchronizations.

4.5. Using concurrent reflective languages

Reflective languages can observe and influence acceptance of message at a meta-level. At the meta-level the full expression-power of the language is available, and therefore virtually any synchronization scheme can be programmed. Examples are ABCLR [35], Actalk [12] and MAUD [17].

Reflective architectures can realise both single-server synchronizations, through individual-based reflection, or multiple-server synchronizations, through group-wide reflection, or a combination of these, through a hybrid architecture [28]. Similar to the languages that provide explicit modules for coordinated behavior, the basic motivation of these languages is to provide open language structures to implement, for example, coordinated behavior among objects. These languages, however, do not aim at providing abstraction mechanisms to express composable multi-server synchronizations.

5. Addressing multi-server synchronization composition problems

5.1. Synchronization in the composition-filters model

In this section, first we will present an object-oriented model to express synchronization compositions. This model is based on the composition-filters concept. Secondly, we will apply this model to solve the multi-server synchronization problems which were presented in the previous section.

The composition-filters model is a modular extension of the conventional object model. Each message that arrives at an object is subject to evaluation and manipulation by the filters of the object. In figure 9, the manipulation of messages by the filters of an object is depicted. One of the things that filters can do, as will be explained in more detail next, is to block a message and put it in the message queue until a certain constraint is satisfied. Filters are modular extensions to the conventional object model because the conventional object model can be viewed as an object with a transparent filter.

![Figure 9. The manipulation and buffering of messages at the filters of an object.](image)

Figure 10 shows the interface definition of class BoundedBuffer, expressed in the Sina syntax. Sina is an experimental language that adopts the composition-filters model. The composition-filters model, however, is general and can be applied to other object-oriented languages as well.

In lines 6 and 7, two methods put and get are declared. The major extensions to the conventional object model are condition and filter declarations, as highlighted by gray boxes. Lines 3 and 4 of the class

---

3 The composition-filters approach is a modeling paradigm rather than the definition of a particular language with fixed semantics. The semantics of objects expressed in the composition-filters model can be largely determined by the semantics of the filters. Similar to extensions of a photo camera, the design of each filter can deal with a different problem and may adopt a different technique. For example, designing a color filter may require different skills than designing an infra-red viewer. Several different filter types have been defined in the past. For example, [2] illustrated how both inheritance and delegation can be simulated using filters. In [3] filters were introduced for defining reusable transactions. Language-adjacent problems were addressed in [4]. In [5], filters were used to abstract coordinated behavior among objects. The application of composition-filters for composing synchronization and real-time specifications was published in [6, 10].

4 Currently, we are developing filter extensions for Smalltalk and Java objects.
A filter of class Wait has the following intuitive definition. When a message arrives at a Wait filter, it can only proceed when the message is accepted by the filter, and it will be blocked otherwise, until the message can be accepted by the filter. In line 9, an input filter called bufferSync of class Wait is declared as:

```
bufferSync : Wait = { Empty=>put,
Partial=>{put, get}, Full=>get };
```

This filter specification corresponds to a mapping from the conditions Empty, Partial and Full to the messages put and get. The ‘\$‘ symbol accepts the messages on its right hand side when -one of- the conditions on the left hand side is satisfied. This mapping results in a guard-based specification, where a guard that is composed of one or more conditions is defined for each message. When new methods are added to the class or one of its subclasses, additional messages can be associated with the conditions, or new conditions can be associated with messages.

The Dispatch filter defines a simple matching mechanism that maps all received messages to the local ('inner') methods. The wild card ‘*‘ implies that all message selectors that are supported by the object are acceptable. The Dispatch filter takes care that a message that does match will be dispatched. Dispatching to a local method results in execution of that method.

### 5.2. Using filters for synchronization inheritance

To demonstrate applicability of the filter concept to the composition of synchronization through inheritance, an additional class with synchronization behavior is introduced. This is class Locking, which implements the mechanisms for locking an object so that all messages it receives except the unlock message are blocked. If the object is unlocked and another unlock message is received, this will be blocked until the object is locked again.

```
<table>
<thead>
<tr>
<th></th>
<th>number</th>
<th>class</th>
<th>interface</th>
<th>comment</th>
<th>conditions</th>
<th>methods</th>
<th>inputfilters</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td></td>
<td>Locking</td>
<td></td>
<td>implement an abstract class</td>
<td>Unlocked; Locked;</td>
<td>lock returns Nil;</td>
<td>lockSync : Wait = { Locked=&gt;unlock,</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td></td>
<td></td>
<td></td>
<td>that can be put into a</td>
<td></td>
<td>unlock returns Nil;</td>
<td>Unlocked=&gt;lock};</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td></td>
<td></td>
<td></td>
<td>locking state and unlocked</td>
<td></td>
<td></td>
<td>execute : Dispatch = { * };</td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
<td>again;</td>
<td></td>
<td></td>
<td></td>
<td>(11)</td>
</tr>
<tr>
<td>(5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>end</td>
</tr>
<tr>
<td>(6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 11. Definition of class Locking.

We omit the implementation of this class that maintains the state (free or locked) of the object, as this is straightforward. In line 4, the conditions Unlocked and Locked are declared. The two methods of the class lock and unlock are declared in lines 6 and 7. As shown in line 9, the synchronization filter lockSync consists of two elements. The first element Locked=>unlock specifies that, if the object is in the Locked state only the unlock message is accepted. The second element is defined using the symbol ‘\$‘ instead of ‘\$=‘. Unlocked=>lock specifies that if the object is unlocked, all messages except the lock message are acceptable.

The composition of the locking synchronization with the bounded buffer synchronization is shown by class LockingBuffer in figure 12:

---

5 In addition to input filters, the composition-filters model also supports output filters. Output filters affect outgoing messages, whereas input filters affect incoming messages. For brevity, we do not further consider output filters in this paper.

6 The current version of the Sina language provides a number of primitive filters types, such as Dispatch, Wait, Meta and Atomic filters and is explained in this paper. The Error filter is similar to the Dispatch filter but it does not provide a method dispatch; it raises an error condition if a message does not pass through the filter [4]. The RealTime filter is used to impose real-time constraints on messages [6]. All filter types can be used for both input and output filters. An important feature of all these filters is that their functionality is orthogonal and therefore they can be combined freely.
The implementation of recommended to make sure that future extensions can still remaining messages that can match the signature of resolves possible naming conflicts. The filter specification can be tailored to select particular messages from either of

The synchronization constraints are encapsulated within the object. Lines 8 and 9 define two filter specifications locker.lockSync and buffer.bufferSync which refer to the wait filters of the internal locker and buffer objects, respectively. These filters ensure that both synchronization constraints are satisfied for a message before it can be executed.

The Dispatch filter in line 10 specifies that all messages that can match the signature of buffer of class BoundedBuffer are dispatched to the buffer object, and all remaining messages that can match the signature of locker of class Locking are dispatched to the locker object. A message that cannot match with either of these will cause an exception. As a result of this filter, class LockingBuffer offers a behavior that is a composition of both BoundedBuffer and Locking. The synchronization for LockingBuffer is a composition of the synchronization of these two classes, and the messages of both classes are available on the interface of LockingBuffer. Declaring internal objects and making their methods available at the interface of the encapsulating object is equivalent to a multiple inheritance mechanism; the classes of these two internals are then the super classes.

Two important properties of the specification of the Dispatch filter are that (a) it puts an ordering on the messages provided by the composed objects, which resolves possible naming conflicts. The filter specification can be tailored to select particular messages from either of the components. And (b), a dispatch of a message to a component differs from a message invocation in that the receiver of the dispatch message is a part of the extended identity of the composed object. The pseudo-variable server will always refer to the original receiver of a message. In summary, the dispatch filter can simulate multiple inheritance while resolving potential name conflicts.

5.3. Using filters for synchronization delegation

The composition of synchronization through delegation is very similar to the previous example; the most important difference is that the class defines a composition using a shared object rather than a private instance. This is exemplified by the SharedLockingBuffer example in figure 13.

In line 3, class SharedLockingBuffer declares an external object sharedLock, which is an instance of class Locking. Object sharedLock is global and can be shared by multiple objects. In line 5, the class also defines an internal object buffer, which is an instance of BoundedBuffer.

In line 7, all the conditions of both buffer and the locking objects are made available for the SharedLockingBuffer object through the wild card symbol "*".

The execute filter dispatches the messages to buffer provided that these messages are supported by its class BoundedBuffer. The messages that are supported by the signature of class Locking are dispatched to the global object sharedLock.
This example demonstrates a combination of inheritance and delegation; the behavior and instance variables of class BoundedBuffer are inherited, and a part of the behavior of SharedLockingBuffer is delegated to an instance of class Locking. In the case of dispatching messages to the sharedLock object, the server pseudo-variable will still refer to the instance of class SharedLockingBuffer, which resolves the so-called self-problem [25]. Obviously, with an appropriate dispatch filter a single object can delegate to multiple objects, even while inheriting from one or more classes.

5.4. Composition of cooperating synchronizations

To exemplify the composition of cooperating synchronizations, we introduce two classes, DistributedBuffer and Distribution. DistributedBuffer cooperates with other distributed buffers to achieve its (synchronization) behavior, whereas class Distribution defines an intermediate object that is responsible for distribution of messages among the buffer objects.

In this specific example, whenever a distributed buffer cannot accept a request, an attempt is made to find another buffer object to service this request. If this attempt does not succeed, the request will be blocked at the original buffer. We will now first describe class DistributedBuffer, and then continue with class Distribution.

Figure 14 shows the definition of DistributedBuffer. In line 3, class DistributedBuffer declares an external object bufferDistribution of class Distribution. Line 5 declares an internal object buffer of class BoundedBuffer. As shown in line 7, all the conditions of the BoundedBuffer object are made available to DistributedBuffer. The distinguishing feature of this class is the filter distributeBuffer, an instance of class Meta which is declared in lines 9 and 10. A meta filter is used to convert messages to objects. This conversion is called reification. If a message is accepted by a meta filter, the message is reified and sent to the designated target object. If the message is rejected, then it passes to the next filter.

```
(1) class DistributedBuffer interface
(2) externals
(3) bufferDistribution: Distribution;
(4) internals
(5) buffer : BoundedBuffer;
(6) conditions
(7) buffer.*;
// conditions of BoundedBuffer also available on
the interface of DistributedBuffer
(8) inputfilters
(9) distributeBuffer : Meta=
 { Full => [put] bufferDistribution.distribute,
 Empty => [get] bufferDistribution.distribute;
 execute:Dispatch={ buffer.* };
 end;
(11) end;
(13) class DistributedBuffer implementation
(14) initial begin
bufferDistributor.putSite(self) end;
// update bufferDistributor to add myself
(15) end;
```

Figure 14. Interface of class DistributedBuffer.

In line 9, if the condition Full evaluates to true, and the request is put, then the message is reified and passed as an argument to object bufferDistribute by invoking the method distribute. This is equivalent to sending the message:

```plaintext
bufferDistribute.distribute(aMessage)
```

Here, bufferDistribute is the external object and aMessage is an instance of class Message which represents the reified message. The method distribute will take care of distributing this request, as will be explained later.

In line 10 the second part of the meta filter is defined. If the condition Empty evaluates to true, and the request is a get message, then this message is reified and passed as an argument to object bufferDistribute by invoking the method distribute. The messages that are not accepted by the meta filter are passed to the next filter, which is a dispatch filter in this case.

Now consider the definition of class Distribution which schedules requests to a pool of server objects. If a service request is received by an instance of Distribution, it forwards this request to one of the server objects in its pool. If the selected server cannot service the request, then the next object in the pool is tried. Distribution of a message is terminated if the message can be accepted by one of the server objects, or all the server objects are tried once for that message. If none of the objects can serve the request, then the request must be handled by the first server. This will usually block the request until it can be handled by that server.
The method which is represented as its argument, to a set of objects.

If needed, a message object can object that dispatches the messages to the internal object of class filter following the meta filter. This is the dispatch filter.

5.5. Polymorphically composing synchronizations

This example demonstrates that multiple synchronization specifications can be composed polymorphically in different contexts. The class AtomicDistributedLocking combines the distributed synchronization that was used previously for bounded buffers with class Locking. At the same time an additional synchronization is composed which enforces atomic execution of this distributed locking.

The first filter in this example, atomic, turns the messages that it accepts into an atomic transaction, which means that the execution of the messages becomes indivisible; either the execution is completely successful (commit), or it is fully abandoned (abort).

One possible application of such a distributed locking algorithm is for locking a limited number of shared resources; whenever a lock request is issued, it is tried to find a currently unlocked server object, which will then be blocked. It is obvious that in the case of multiple concurrent requests, atomicity of this distributed locking algorithm is important to ensure consistency.
with a specification that turns the involved messages into atomic transactions.

5.6. Realizing ordered compositions

The example we use to demonstrate ordered compositions is class ProtectedLockingBuffer, which composes properties of class Login, class BoundedBuffer and class Locking:

```
(1) class ProtectedLockingBuffer interface
(2) externals
(3) verifier : Login;
(4) internals
(5) buffer : BoundedBuffer;
(6) locker : Locking;
(7) input filters
(8) order : Error = { verifier.*; locker.*; buffer.* };
// note the semicolons define serialization
(9) locker.lockSync;
(10) buf.bufferSync;
(11) execute : Dispatch = { verifier.*, buffer.*, locker.* };
(12) end;
```

Figure 18. Interface part of class ProtectedLockingBuffer.

The filter specifications in lines 8-11 are again the crucial part of this class definition: the Error filter order in line 8 specifies ordering. This is achieved by specifying a pattern of messages, separated by semicolons: "{ verifier.*; locker.*; buffer.* }". A sequence of messages m1, m2, m3, .. that the object receives will only pass this filter if, and only if, the messages are supported by respectively class Login (m1, m4, m7, ..), BoundedBuffer (m2, m5, m8, ..) and Locking (m3, m6, m9, ..). This means that the filter object maintains an internal state, designating which category of messages is expected next. The messages that are acceptable, simply pass to the next filter unmodified. Others will raise an error condition.

The two next filters (lines 9-10) specify composition of synchronization of class ProtectedLockingBuffer from the Locking and BoundedBuffer class, respectively. Finally, the dispatch filter execute in line 11 composes the behavior of the ProtectedLockingBuffer from the three existing classes. The important thing to notice is that the composition of these different aspects is completely orthogonal, in the sense that the specification of one synchronization property does not prohibit the specification of another.

6. Evaluation and conclusions

Most existing object-oriented concurrent languages do not provide composable synchronization mechanisms among multiple server objects. We think that object-oriented languages must support both single and multiple-server synchronizations among objects in a composable way.

The contribution of this paper is twofold. First, a new framework is introduced for the classification of the multi-server synchronization problems. Second, an object-oriented model is defined to express these synchronization problems in a composable way. The problems of expressing multi-server synchronizations were explained through a set of related examples. The same examples were also used to illustrate the applicability of the composition-filters concept.

As explained in section 5, the composition-filters model provides modular and orthogonal extensions to the conventional object model. Various filters can be defined to address different issues. For example, in figure 10, in the definition of class BoundedBuffer, a wait filter was used to express synchronization.

By adopting multiple filters, one can compose synchronization schemes of several objects. For example, in figure 12, the synchronization schemes of BoundedBuffer and Locking were composed by cascading their filters at the interface of class LockingBuffer. Similarly, since an object's interface can be composed from internal and/or external objects, both inheritance and delegation mechanisms can be specified. In the definition of SharedLockingBuffer, figure 13 illustrates how synchronization distribution through delegation can be implemented using composition filters.

Meta-filters can reify messages and pass them to a meta-level object. Since a meta-filter can be composed with a wait filter, a meta-level behavior such as distribution, can be composed with object's synchronization specification. This feature can be used to specify cooperating synchronizations. The definition of DistributedBuffer in figure 14 takes this approach.

An atomic filter can mark the received messages as atomic. After passing this filter, a message is executed as an atomic thread. The method that is activated through such a message executes as an atomic transaction. This mechanism allows the atomic transaction semantics to be polymorphically composed with a set of messages, since a filter may allow different messages to pass through it. Polymorphic composition of synchronizations (serialization in this case), is illustrated by AtomicDistributedLocking in figure 17.

Finally, since filters are message processors, they can order the messages that can be accepted by an object. This mechanism is useful in expressing ordered compositions, such as ProtectedLockingBuffer in figure 18.

In the application of the composition-filters model to multi-server synchronization, we are currently
investigating a number of open problems. For example, cooperation among functionally compatible but procedurally incompatible synchronizations is not trivial. One server object, for instance, can adopt two-phase locking for serialization, and the other object may adopt an optimistic approach [19].

References

[14] Corradi & L. Leonardi, PO constraints as tools to synchronize active objects, JOOP October 91, pp. 41-53


