A Synchronization Mechanism for an Object Oriented Distributed System

D. Decouchant, P. Le Dot, M. Riveill, C. Roisin, X. Rousset de Pina

Bull - IMAG / Systèmes
Z. I. de Mayencin - 2, rue Vignate - 38610 Gières - France
e-mail: rousset@imag.fr

Abstract

This paper presents a mechanism for synchronizing shared objects in a distributed system based on persistent, typed objects. Synchronization constraints are expressed as separate control clauses and are factorized for a class of objects. The interference of this mechanism with inheritance and transactions is examined and solutions are proposed. Examples of synchronized objects are provided and a semaphore-based implementation of the mechanism is described.
Keywords: synchronization, distributed system, object-oriented programming.

1. INTRODUCTION

A major characteristic of many computer systems is *data sharing* by a number of concurrent activities. Concurrent access to shared data must be controlled in order to guarantee that the data remain in a consistent state. This problem has been known for a long time, and a number of solutions have been proposed.

However, two new features have appeared in recent systems:

- *object orientation*, i.e. a specific mode of information structuring, mainly characterized by the association of operations to data, by dynamic instance creation and by inheritance of properties,
- *distribution*, i.e. the ability to execute computations spanning a number of physical nodes on a network, and to move objects from one node to another.

Few synchronization mechanisms have been proposed for object-oriented systems, the main reason being that few such systems support concurrency and shared objects. In Emerald [1], the only synchronization tool provided is a "monitor" construct allowing the encapsulation of data and methods with exclusive access. Trellis/Owl [12] proposes a similar tool: locks and wait queues. In Hybrid [13], delay queues allow scheduling activities in the same way as condition variables in monitors. All these mechanisms are limited and cannot easily express canonical synchronization schemes.

Many synchronization tools have been proposed for modular or distributed environments and an exhaustive overview is available in [7]. These mechanisms include monitors [8], constructs based on synchronous message passing and guarded commands [6], such as DP [2] or CSP [9], and path expressions [3]. These mechanisms are powerful but difficult to implement especially in a distributed environment. In some implementations such as Modula2 monitors [15], mutual excusion is the only available synchronization construct. Other implementations provide locks for each method or group of methods.

One of the most attractive tools for the domain we are interested in, is the mediators mechanism [7]: it is based on guarded commands with statements allowing mutually exclusive service calls (the *exec* statement), or asynchronous calls (the *spawn* statement), and return with the *release* statement. These statements allow the modification of scheduling waiting queues. Guards express pre-conditions that can be programmed using the variables of the mediator, the service requested by the client (name of the service, value of the parameters and key identifying the client) and status about pending requests. This tool is very powerful, but it may be too complex for most application programmers. It also seems difficult to implement in a efficient way. This analysis motivated us to find a mechanism with an expressive power between the locks and the mediators.

The synchronization mechanism presented in this paper has been designed for the object-oriented distributed system Guide, currently being implemented at Bull-IMAG, a joint research unit of Bull and the IMAG Institute. This system also embodies the object-oriented architecture defined in the Comandos project under the ESPRIT Program supported by the Commission of European Communities [10].

The paper is organized as follows. We briefly introduce the main features of the Guide system and language that are relevant to the topic of this paper in Section 2. The expression of synchronization for shared objects is described in Sections 3 and 4. An integration of this mechanism with inheritance is proposed in Section 5 and the relation with the transaction model is analyzed in Section 6. An implementation of our mechanism is presented in Section 7.

2. OVERVIEW OF GUIDE

An **object** is the association of a set of data (its **state**) and a set of procedures (its **operations** or **methods**), which provide the only means for consulting or changing the state. Objects are typed. A **type** is a specification of the operations applicable to objects of that type, each operation being defined by its

signature. The internal representation of object and the code of methods are not included in the type definition.

The following examples are written in Guide, an object-oriented language that has been developed within the project [11], and the notations should be self-explanatory. The following declaration specifies a type.

```
TYPE ProducerConsumer IS
   METHOD Put(IN Element);
   // deposit an Element in the buffer
   METHOD Get(OUT Element);
   // extract an Element from the buffer
END ProducerConsumer.
// type Element is supposed to have been
// previously defined.
```

A **class** is an object model and an object constructor. It defines a specific implementation of a type and a mechanism for the creation of objects represented according to this implementation. These objects are **instances** of the class. A class provides a description of the internal representation of its instances and the code of the operations applicable to these instances. The operations that create and delete instances are provided by the run-time system. Class instances have the same behavior (described by the type implemented by the class); in addition, they share the code of their operations, although each instance has a separate state representation.

```
CLASS FixedSizeBuffer
IMPLEMENTS ProducerConsumer IS
   CONST size=<some constant>;
   buffer : ARRAY[0..size-1] OF Element;
   first, last: Integer = 0, 0;
   METHOD Put(IN m: Element);
      BEGIN
      buffer[last]:=m;
      last:=last+1 MOD size;
      END Put;
   METHOD Get(OUT m: Element);
      BEGIN
      m:=buffer[first];
      first:= first+1 MOD size;
      END Get;
END FixedSizeBuffer.
```

This class could have been defined as generic, i.e. parameterized by a value or a type, but we retain the non-generic form in this example to make the programs simpler. Note that no synchronization constraints are specified here.

A given type may be implemented by several different classes. The instances of two different classes which implement the same type share the same external interface but have different internal representations and different operation programs.

Objects are **persistent**, i.e. the lifetime of an object is independent of that of the program or process in which the object was created. An object is identified by a system-wide **reference** (a universal unique name). An object is always located on a single node, i.e. objects are not partitioned (however, an object may migrate from one node to another and include references to other, possibly remote, objects).

Objects are essentially passive. The active entities of the system are called jobs and activities. A **job** may be regarded as a multiprocessor distributed virtual machine. It defines an adressing space, possibly distributed on several nodes, in which objects may be dynamically bound upon invocation. A job contains one or more parallel **activities**, i.e. sequential threads of control. The execution of an activity is described as a sequence of synchronous invocations on object methods.

Communication between activities, within the same job or in different jobs, takes place through shared objects. Examples of widely used shared objects are communication channels, whose function is similar to that of Unix pipes; however, they carry typed objects instead of bytes.

3. SYNCHRONIZATION CONSTRAINTS

3.1. Definition

Our basic choice is to express synchronization as a set of constraints associated with **objects**, not as primitives appearing in the program of activities. This is fully consistent with the object approach. In addition, since the class mechanism is used to describe the behavior of objects, the specification of synchronization is included within a class and applies to all instances of the class.

The only way for an activity to read or modify an object is to execute a method of this object. Therefore, we specify synchronization as a set of **activation conditions**, which are similar to Dijkstra's guarded commands: an activation condition is attached to a method, it is calculated within a critical section and it must be satisfied before the execution of this method may start. This condition is expressed as a function of the internal state of the object and of the parameters of the invocation. If no activation condition is attached to a method, then the execution of this method is unconstrained.

The semantics of such a mechanism closely depends on when the guards are evaluated:

- if the guards are evaluated each time the object state is modified, the condition semantics is "as soon as …";
- if the guards are evaluated each time a method has terminated its execution, the semantics is "when…";

For the conceptual point of view the first approach is more accurate but it is time consuming. On the contrary, the "when…" semantics is less time consuming but it must be used carefully because it can lead to deadlocks, resulting not from a programming fault but from the evaluation mechanism itself. The semantics chosen in Guide is closer to the second one, but we will see in Section 7 how our implementation can prevent deadlocks.

3.2. Expressing activation conditions

In our language, the activation conditions are regrouped in a **control clause** at the end of the class description. The syntax of the control clause is as follows:

```
CONTROL
   [<method name> : <activation condition>]*
```

where <activation condition> is a boolean expression which may contain the following elements:

- instance variables which are part of the internal state of an instance,
- actual parameters of the method,
- synchronization counters.

Synchronization counters are internal data that specify, for each method of a given object, the total number of invocations, the total number of completed executions, the current number of pending invocations, etc. These counters are automatically updated by the system. Synchronization based on counters was introduced in another context by Robert and Verjus [14]. The following counters are defined for each method *m*:

invoked(m)	:	number of invocations of method m
started(m)	:	number of accepted (non-blocked) invocations of method m
completed(m)	:	number of completed executions of method m
current(m)	:	number of activities currently executing method m
pending(m)	:	number of activities currently blocked on invocations to m

The following relationships hold :

current(m) = started(m) - completed(m)
pending(m) = invoked(m) - started(m)

The first three counters are created and initialized to 0 when the instance is created. They record total numbers, and are non-decreasing.

3.3. Fixed sized buffer example

Activation conditions are illustrated by the following control clause for the FixedSizeBuffer defined in section 2. The conditions enforce the usual bounded buffer synchronization scheme:

```
CLASS FixedSizeBuffer
IMPLEMENTS ProducerConsumer IS
// internal state variables; see § 2
   METHOD Put(IN m: Element);
   // implementation of method Put; see § 2
   METHOD Get(OUT m: Element);
   // implementation of method Get; see § 2
CONTROL
   Put:(completed(Put) - completed(Get)<size)
      AND current(Put)=0;
   Get:(completed(Put) > completed(Get))
      AND current(Get)=0;
END FixedSizeBuffer.
```

The two statements in the control clause specify activation conditions for methods Put and Get, respectively. The activation condition for Put specifies that no Put is active and the buffer is not full. The activation condition for Get specifies that no Get is active and the buffer is not empty. In these conditions, we can see the use of the internal state of objects (constant size) to express synchronization constraints.

3.4. Expression of usual synchronization scheme

Mutual exclusion between the k methods m_1, m_2, ..., m_k within an object is expressed by the following activation condition : Σ current(m_i) = 0

Since this condition is frequently used, it is replaced the keyword EXCLUSIVE(m_1, m_2, ..., m_k) in the control clause. The language offers similar keywords for expressing other canonical synchronization schemes (such as readers and writers problem).

It should be noted that activation conditions are expressed using only boolean expressions. Modification of internal variables or more general algorithms are not allowed. As a consequence some synchronization schemes can not be expressed directly. This limitation is illustrated in the next section.

4. A MORE COMPLETE EXAMPLE

4.1. "Readers and Writers" problem

Among all canonical synchronization problems which have been programmed using our mechanism, the readers and writers problem best illustrates the power and limitations of the mechanism.

Let us define type Readerswriters which expresses the access methods for objects shared according to the readers and writers scheme.

```
TYPE Readerswriters IS
   METHOD Write (IN e: Element, dep: Integer);
   METHOD Read (IN dep: Integer): Element;
END Readerswriters.
```

4.2. Scheme with priority to readers

The following class implements the Readers and Writers scheme with priority to Readers (Writers are delayed if a Reader or a Writer is working, or if a Reader is pending):

```
CLASS ReadersFirst IMPLEMENTS Readerswriters IS
   <declaration of internal state variables>
   METHOD Write (IN e: Element, dep: Integer);
   // <implementation of method Write>
   METHOD Read (IN dep: Integer): Element;
   // <implementation of method Read>
CONTROL
   Write: ((current(Write) + current(Read)) = 0)
      AND (pending(Read)= 0);
   Read: current(Write) = 0;
END ReadersFirst.
```

4.3. Scheme with priority to writers

In the same manner, we define the class WritersFirst, which gives priority to the writers (e.g. reads are delayed if a write call is current or pending).

```
CLASS WritersFirst IMPLEMENTS Readerswriters IS
// same code as above
CONTROL
   Write: (current(Write) + current(Read)) = 0;
   Read: (current(Write) = 0)
      AND (pending(Write) = 0);
END WritersFirst.
```

4.4. Symmetrical priority scheme for readers and writers

The implementation of a symmetrical scheme is more difficult. In this scheme, a Write request may only be executed after completion of all Reads and Writes prior to it. Conversely, a Read request may only be executed after completion of all Writes prior to it. When a Read is in progress, any incoming Reads are processed until a Write request arrives.

Therefore,we must be able to keep track of pending invocations of Writes during the execution of both primitives. As we have restricted our control clauses to boolean expressions, this is achieved by defining two additional methods, AskRead and AskWrite, whose bodies are empty and whose control clauses are used to count the number of pending Writes and pending Reads.

This can be described using subtyping and inheritance. The relation between synchronization and inheritance will be studied in the next section.

```
TYPE TNoPriority SUBTYPE OF Readerswriters IS
   METHOD AskRead (IN Integer);
   // Methods Read and Write are inherited
   METHOD AskWrite (IN Integer);
   // from the supertype Readerswriters.
END TNoPriority.

CLASS NoPriority  SUBCLASS OF Readersfirst
                  IMPLEMENTS TNoPriority IS
   METHOD AskWrite (IN
            NbWrite_NbReadBeforeMe: Integer);
      BEGIN
      // EMPTY METHOD
      // used only to keep track of the
      // invoked(Write)+invoked(Read)
      // value at call time.
      END AskWrite;
   METHOD AskRead (IN NbWriteBeforeMe: Integer);
      BEGIN
      // EMPTY METHOD
      // used only to keep track of the
      // invoked(Write) value at call time.
      END AskRead;
   METHOD Write (IN e: Element, dep: Integer);
      x : Integer ;
      BEGIN
      x := invoked(Write)+invoked(Read) - 1;
      SELF.AskWrite(x);
      // control the Write access
      SUPER.Write(e,dep);
      // call the method Write of the SuperClass
      END Write;
   METHOD Read (IN dep: Integer): Element;
      x : Integer ;
      BEGIN
      x := invoked(Write) ;
      SELF.AskRead(x);
      // to control the Read access
      RETURN SUPER.Read(dep);
      // call the method Read of the SuperClass
      END Read;
CONTROL
   AskWrite: (completed(Write) + completed(Read))
      = NbWrite_NbReadBeforeMe;
   AskRead: completed(Write) = NbWriteBeforeMe;
END NoPriority.
```

It can be noted that no activation condition is associated with Read and Write methods: the control is performed when methods AskRead and AskWrite are called.

In fact, this solution is incorrect because the calculus of NbWrite_NbReadBeforeMe within the method Write is not done in mutual exclusion. The two counters involved in this calculus are directly modified by the kernel, that may lead to an inconsistent evaluation of `NbWrite_NbReadBeforeMe`. Thus it is necessary to add to the state of the objects a copy of these synchronization counters, and to define another method which will modify each of these copies in an exclusive manner. We give below the new statements which are added to the NoPriority class.

```
// state of class NoPriority
nbRead : Integer = 0;
// synchronization counter for
// Read accesses
nbWrite : Integer = 0;
// synchronization counter for
// Write accesses

// added method for which
// exclusive access is defined
METHOD GetOrder (IN read : Boolean) : Integer;
BEGIN
   IF read THEN
      nbRead := nbRead + 1;
      RETURN nbWrite
   ELSE
      nbWrite := nbWrite + 1;
      RETURN nbWrite + nbRead - 1;
   END;
END GetOrder;
```

The code of the other methods of the class remains unchanged, except a call for the method GetOrder at the beginning of methods Write and Read. The first instructions of the method Write are:

```
x := SELF.GetOrder(FALSE) ;
SELF.AskWrite(x);
// control the Write access
```

while we have for the method Read:

```
x := SELF.GetOrder(TRUE) ;
SELF.AskRead(x);
// to control the Read access
```

In the CONTROL section the exclusion condition for the method GetOrder is added, i.e.:

```
GetOrder: current(GetOrder) = 0;
// exclusive method
```

This example shows that complex synchronization schemes require the definition of dedicated methods to compute in mutual exclusion synchronization constraints (i.e. in the given example, the number of operations read and write which must have been performed before a specific read or write can begin). Similar problems arise in other synchronization mechanisms; for example, in the path expression scheme [3], the introduction of void functions is often necessary to express synchronization constraints.

4.5. Proposals for the language level

The examples given above have shown that the proposed mechanism is powerful enough to express the canonical

synchronization schemes. For simple schemes, its use is very easy, however for more complex ones, the programmer has to overcome the intrinsic difficulty of the algorithm.

In most applications, synchronization schemes are usually standard and simple : exclusive access, sharing with readers or writers rights. In our language, syntactic shortcuts are given to avoid using the synchronization counters in simple cases.

5. SYNCHRONIZATION AND INHERITANCE

5.1. Presentation of the problem

Synchronization counters provide a simple and short way for the expression of synchronization constraints on shared objects. This section examines the relationship between the synchronization model and the inheritance mechanism. The object model defined in Guide provides single inheritance, with possible overloading. For example, let M be a method defined in class C and let C1 be a subclass of C. Then the program of M may be overloaded (i.e. redefined) in C1. However, the program of the original method M, as defined in C, is still accessible within C1 using keyword SUPER, (although it is not accessible by users of instances of C1, who only see the overloaded method).

We specify below how the inheritance mechanism applies to synchronization clauses. Two issues should be considered:

- for inherited methods, is the activation condition inherited too or is it redefined ?
- for overloaded methods, is the original method of the superclass (accessible in the subclass with SUPER) considered as a synchronized method in the subclass or not ?

Several solutions to the first issue seem to be consistent with the object approach, but some are not useful for programming. For example, enforcing systematic inheritance of synchronization conditions for each inherited method is not appropriate because it is sometimes necessary to define a new condition which takes into account the new method of the subclass. A reasonable solution for that point could be: implicit inheritance of activation conditions for all methods of the superclass, excepted when new conditions are required for some methods in the subclass; in this case, new conditions are be defined for these methods and replace the previous conditions. Usually, the overloaded conditions are stronger than the corresponding ones of the superclass but this is not statically checked by the compiler.

For the second problem mentioned here, our experience of programming with this tool (see examples of section 4 above) convinced us that we don't need to synchronize SUPER calls. The overloaded method in which the SUPER call is made is usually synchronized with a condition stronger or equal than the one defined in the superclass. Moreover, in most cases the evaluation of the condition before the execution of the SUPER call would lead to a deadlock (see example in section 4.3). And if we consider that the methods which have been overloaded in a subclass are not part of the subclass interface, they do not have to be synchronized. A method called using the keyword SUPER is equivalent to an internal procedure call whose code is defined in the superclass.

5.2. Implementation

The following table summarizes the various interferences between synchronization and inheritance, and for each of them, gives the solution we have implemented.

statement	synchronization processing
SUPER.op (…)	nothing.
ref.op (…), op is not inherited	evaluation of the synchronization condition of method op.
ref.op (…) op is inherited or overloaded	evaluation of the synchronization condition of method op; this condition is either the one which has been specified in the current class (overloaded condition), or the one of the inherited class (inherited condition).

5.3. Example

The readers/writers problem is considered again to illustrate how synchronization conditions of inherited classes are used. We firstly define a general class in which the activation conditions express only a consistent access control. This class will be inherited to define classes for different readers and writers access policies.

```
CLASS SyncReaderswriters
IMPLEMENTS Readerswriters IS
<declarations of internal state variables>
   METHOD Write (IN e: Element, dep: Integer);
      <implementation of method Write>
   METHOD Read (IN dep: Integer): Element;
      <implementation of method Read>
CONTROL
   Write: current(Read) =0 AND current(Write) =0;
   Read: current(Write) = 0;
END SyncReaderswriters.
```

The class allows the creation of objects on which access is controlled to insure consistency. Other access restrictions are implemented according to different Readers and Writers policies (ReadersFirst, WritersFirst, …), using inherited types and classes to specify additional conditions and operations.

```
CLASS ReadersFirst
SUBCLASS OF SyncReadersWriters
IMPLEMENTS Readerswriters IS
   // methods Write and Read are inherited
CONTROL
   Write: current(Read) + current(Write) = 0
      AND pending(Read) = 0;
      // new condition for the Write method
      // but inherited condition for
      // the Read method
END ReadersFirst.
```

6. SYNCHRONIZATION AND TRANSACTIONS

In the transactional model chosen for Guide, a transaction is a sequence of methods calls of objects bracketed by a pair of BEGIN-TRANSACTION and COMMIT-TRANSACTION. The serialization property is insured by a two-phase locking

mechanism. In the first implementation locking is simply based on read and write locks. The use of typed locks associated with methods will be considered in the future.

Used independently, the transaction locking or synchronization mechanisms can lead to data inconsistencies or deadlocks. Let us for example define a class Stock with the two synchronized methods Produce and Consume:

```
CLASS Stock IMPLEMENTS Stock IS
   empty, full: Boolean;
<declaration of the remainder of the state>
   METHOD Produce (IN Item);
      <implementation of method Produce>
   METHOD Consume (OUT Item) ;
      <implementation of method Consume>
CONTROL
   Produce: full = FALSE;
   Consume: empty = FALSE;
END Stock.
```

An activity which wants to consume when the stock is empty has to wait for the production of an item (the boolean empty will be set to FALSE). If this occurs inside a transaction, the problem is more complex: if the locks (which are exclusive locks in this case) are acquired before evaluating synchronization conditions, we can obtain the following situation:

current state of the stock: empty = TRUE.

T1: transaction which wants to consume	T2: transaction which wants to produce
T1 gets an exclusive lock but is blocked by the synchronization constraint (empty is TRUE).	T2 asks for an exclusive lock, T1 has it, thus T2 waits and can't produce to enable the consumer.

------> deadlock situation.

In the reverse manner, if the synchronization condition is evaluated before getting a blocking lock, the condition could have become false when the lock is acquired: the object is accessed whereas the synchronization condition is false.

To avoid the former problem, the locking and the synchronisation conditions must behave like a global condition (i.e. the lock is set and the method is run only when both conditions are true). When a method ends, the synchronization counters are updated but the locks have to be kept until the end of the transaction.

7. IMPLEMENTATION

This section describes the principle of the implementation of the synchronization mechanism presented in the previous sections. This description uses semaphores to specify the scheduling of low-level processes, although any equivalent mechanism would be convenient.

First let us briefly describe how distribution is managed. A Guide activity is represented by several processes, one per node visited by the activity. Remember that an object is located on a single node. Using the object reference, a locating algorithm provides the activity with the object current location. The calling activity extends to the site of the called object, or, conversely, the object is moved to the site where the call was executed, depending on a load sharing policy. Extension of an activity to a node essentially implies the creation of a new process on that node, as a representative for the calling activity (if a representative is already present, it is reused), followed by a remote invocation using that representative. To summarize, the synchronization is always performed locally on a node and involves the local processes which represent the calling activities on that node.

7.1. Principle of the implementation

The current implementation of the GUIDE system is carried out on top of UNIX. Activities are mapped onto UNIX processes and shared objects are mapped in shared memory segments: implementation details are described in [4].

The invocation of a synchronized object method is bracketed by calls to a Prologue and an Epilogue, taking as argument the index of the method. The calls are automatically inserted by the compiler. The function of the Prologue is to evaluate the activation condition associated with the method, and to block the invoking activity if necessary. The function of the Epilogue is to wake up the activities which are currently waiting for an activation condition.

Each activity has a private semaphore used to block its execution while waiting for a condition to be satisfied. Since an activity is essentially a sequence of synchronous method invocations, a single semaphore per activity is sufficient.

This algorithm uses the following data structures:

a) for each method of a synchronized object:

- A field recording the current counter values. This field is part of the object state.
- An internal boolean function (named Method-SynchroFunction in the previous algorithm) which evaluates the activation condition of the method and modifies the associated counters.

These structures are automatically generated by the compiler as a result of the parsing of control clauses.

b) an activity queue, global to each node, which records for each accessed object, the identifier of the activities which are waiting for execution of a method on this object.

c) a mutual exclusion semaphore (named Synchro_Eval), one per node, ensuring:

- atomic evaluation of the synchronization functions,
- atomic modification of the waiting activities list.

The algorithms of the Prologue and the Epilogue are the following:

```
  /* PROLOGUE: evaluation of the synchronization
                    expression */
P (Synchro_Eval);
WHILE (NOT MethodSynchroFunction(object, method))
DO
   /* Evaluate the control clause associated
   with the effective method code */
   BEGIN
      /* Activity must wait on this object:
         1- insert its identification into
```

```
            the waiting activity list for
            this object,
         2- release the CPU. */
      Wait_On_This_Object (object);
      V (Synchro_Eval);
      StopThisActivity ();
      P (Synchro_Eval);
   END
   V (Synchro_Eval);

   /* EFFECTIVE CALL */
   EffectiveMethodCall ();

   /* EPILOGUE: restart of
   stopped activities */
   P (Synchro_Eval);
   activity=FirstWaitingActivityOnObject(object);
   WHILE ( activity != NIL_ACTIVITY ) DO
   /* All waiting activities must be resumed
because they must themselves re-evaluate their
own synchronization expressions. Several resumed
activities may actually be restarted, while the
others will be waiting again */
   BEGIN
      Resume_Activity (activity);
      activity =
      NextWaitingActivityOnObject (object);
   END
   V (Synchro_Eval);
```

The first implementation of this mechanism was defined using UNIX semaphores. For more efficiency, UNIX semaphores have been replaced by specific drivers. More details about this implementation can be found in [5].

7.2. Implementation deadlocks

In this section, we discuss possible deadlocks which can be generated by our synchronization implementation (and which are not the consequence of a programming error). It is important to note that synchronization conditions are expressed in terms of method invocations (cf. section 4.1), and of object state variables. As previously noticed, the underlying environment does not allow any implementation of a synchronization mechanism which insures that each activity will be restarted as soon as its current method synchronization expression becomes true. Such an implementation would be too expensive in terms of generated code and in terms of context switching. Our goal is to propose a mechanism in which the evaluation of the synchronization expressions is only performed by the kernel at specific points. As presented in the synchronization algorithm shown in the previous section, these points are included in the prologue and epilogue parts. But these basic points are not sufficient to avoid deadlocks.

The following example shows possible deadlocks generated by this implementation.

```
CLASS Example IMPLEMENTS Example_Type IS
   METHOD M1;
   // <implementation of method M1>
   METHOD M2;
   BEGIN
      O2.M3;
   END M2;
CONTROL
   M1 : invoked (M2) >= 1;
END Example.
```

Assuming that two activities execute successively O1.M1 and O1.M2 instructions (O1 is an instance of class Example), the first activity will block on the synchronization function associated with M1. Now, if we suppose that O2.M3 method call blocks, then the two activities are blocked. However, the value of the "invoked" counter associated with M2 is now 1, and it will be possible to resume the execution of the first activity. This is a deadlock caused by the implementation of the synchronization mechanism, not by user design: waiting activities are only restarted when an activity exits from a method, and not when the instances variables (and the counters) are modified.

The solution we have adopted is the following: if a method of a synchronized object modifies any variable involved in a synchronization condition of its class, the compiler generates a call to the epilogue before every invocation done inside the method. So, even if the invocation blocks the activity which is executing it, one of the activity blocked on the invocking object may restart.

8. CONCLUSION

We have presented a synchronization mechanism for shared typed objects in a distributed environment. This mechanism is characterized by the following features:

- Synchronization constraints are common to a class of objects, and are expressed as separate control clauses in the program of that class. A control clause contains an activation condition (possibly empty) for each method defined in the class.
- An activation condition associated with a method specifies a "guard" that must be true before the execution of the method may start. This condition is expressed using the internal state of the object, the parameters of the method invocation, and internal counters attached to the object.
- Synchronization constraints are inherited by subclassing mechanism and can be overloaded.

We have experimented this synchronization tool for applications involving synchronized access to their components. For example, we have implemented general-purpose services shared between system users such as a name service, and several medium-sized experimental applications (a mailer, an electronic agenda, a bibliography manager, etc). With the synchronization mechanism described in this paper, we got no particuliar difficulties to express the different control schemes of these applications.

The choice of a basic synchronization mechanism is obviously a tradeoff between complexity and power. From the point of view of the expression power, execution conditions may be viewed as a subset of the mediator concept. However, we think that this basic tool is satisfactory for the following reasons:

- Most synchronization schemes can be expressed with execution conditions, and basic schemes are directly

available at the language level (EXCLUSIVE, READER, WRITER).

- We think that the system should only provide basic mechanisms. The design of more complex synchronization schemes is under the responsibility of the application designer.

Acknowledgments

We would like to thank Roland Balter and Jacques Mossière for their contribution to the definition of the synchronization mechanism and Sacha Krakowiak for his help reviewing this paper. Interaction with members of the Object-Oriented group of the Comandos ESPRIT project was beneficial to this work. The Guide project is also partially supported by Centre National de la Recherche Scientifique (CNRS) and Comandos project under the ESPRIT Program supported by the Commission of European Communities.

References

[1] A.P. Black, N. Hutchinson, E. Jul, H. Levy, L. Carter, Distribution and abstract types in Emerald, *IEEE Trans. on Software Engineering*, vol SE-12, dec. 1986.

[2] P. Brinch Hansen, Distributed Processes: a concurrent programming concept, *CACM* vol.21, n°11, nov. 1978, pp.934-941.

[3] Campbell R.H., Habermann A.N., The specification of process synchronization by path expressions, *Operating Systems*, Lecture Notes in Computer Science, Springer-Verlag (1974), pp. 89-102.

[4] D. Decouchant, A. Duda, A. Freyssinet, E. Paire, M. Riveill, X. Rousset de Pina, G. Vandôme, GUIDE: an implemention of the Comandos object oriented architecture on Unix, *Proceedings of EUUG Autumn Conference*, Lisbon, oct. 1988, pp. 181-193.

[5] D. Decouchant, E. Paire, M. Riveill, Efficient implementation of low-level synchronization primitives in the UNIX-based GUIDE kernel, *Proceedings of EUUG Autumn conference*, Vienna, sept. 1989.

[6] E.W. Dijkstra, Guarded commands. Non-deternimacy and formal derivation of programs, *CACM* vol.18 n°8, Aug. 1975, pp.453-457.

[7] J.E. Grass, R.H. Campbell, Mediators: a synchronization mechanism, *Proceedings of 6t^h International Conference on Distributed Computing Sytems*, Cambridge (1986), pp 468-477.

[8] C.A.R. Hoare, Monitors: an operating system structuring concept, *CACM* vol.17, n°10, oct. 1978, pp.549-557.

[9] C.A.R. Hoare, Communicating Sequential Processes, *CACM* vol.21, n°8, aug. 1978 pp.666-677.

[10] C.J. Horn, S. Krakowiak, An object-oriented architecture for distributed office systems, *Proc. ESPRIT Technical Conf.*, Brussels, sept. 1987.

[11] S. Krakowiak, M. Meysembourg, M. Riveill, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina: Design and implementation of an Object-Oriented, strongly typed language for distributed applications, *Journal of Object Oriented Programming* vol. 3, n° 3, sep /oct 1990, pp 11-22.

[12] J. Moss, W. Kohler, Concurrency features for the Trellis/Owl language, *proceedings of ECOOP*, 1987.

[13] O.M. Nierstrasz, Active objects in Hybrid, *Proceedings of OOPSLA'87*, vol 22, ACM SIGPLAN Notices n°12, Dec. 1987, pp. 243-253.

[14] P. Robert, J.-P. Verjus, Toward autonomous descriptions of synchronization modules, *Proceedings of IFIP. Congress* (B. Gilchrist, ed.), North-Holland (1977), pp. 981-986.

[15] N. Wirth, Programming in Modula-2, *Springer-Verlag*, New-York, 1982.