SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 19(3), 235–256 (MARCH 1989)

An Implementation of the Object-oriented Concurrent Programming Language SINA

ANAND TRIPATHI AND ERIC BERGE
Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455, U.S.A.

AND

MEHMET AKSIT
Department of Computer Science, University of Twente, 7500 AE Enschede, The Netherlands

SUMMARY

SINA is an object-oriented language for distributed and concurrent programming. The primary focus of this paper is on the object-oriented concurrent programming mechanisms of SINA and their implementation. This paper presents the SINA constructs for concurrent programming and inter-object communication, some illustrative examples and a message-based implementation model for SINA that we have used in our current implementation.

KEY WORDS Object-oriented programming Distributed computing Concurrent programming

INTRODUCTION

SINA[1] is an object-oriented language for concurrent and distributed programming. SINA is an experimental language in the sense that it has been designed and implemented to study issues in object-oriented concurrent programming such as data abstraction, concurrency, synchronization, inter-object communication, inheritance, reuseability and delegation. The primary focus of this paper is on the implementation of SINA constructs to support concurrency and synchronization within objects, and inter-object communication. This paper is not intended to give the complete definition of this language. Several other object-oriented programming features of this language, such as inheritance, delegation and reuseability, have been implemented and evaluated in a separate project.[2]

The object-oriented approach to system design closely matches our model of the reality, whereas the procedural or functional languages concentrate on operational and algorithmic abstractions the problem.[3] Since 'real-world' objects are characterized by both inter-object and intra-object concurrency, object-oriented programming languages that support concurrency would be more expressive. Most object-oriented languages today do not support concurrency. Languages such as Smalltalk-80,[4] Objective-C,[5]

0038–0644/89/030235–22$11.00

Received 22 February 1988
Revised 29 September 1988

C++,[6] Trellis/Owl,[7] CommonLoops[8] and CommonObjects[9] aim at high reuseability, without emphasizing the need for concurrency. Examples of object-oriented languages that provide mechanisms for concurrency are Orient84/K,[10] ABCL/1,[11] Concurrent Smalltalk,[12, 13] Hybrid[14] and Pool-T.[15] Most of these languages, however, do not support concurrency within objects.

SINA supports concurrency within objects, since SINA objects can encapsulate other active objects and multiple concurrent processes. A SINA object can encapsulate a shared resource as well as its related synchronization mechanisms together in the same module. This alleviates many of the problems that arise in using nested monitor structures[16–18] for hierarchically structured resource management systems. Another interesting feature is SINA's capability-based inter-object communication, which allows communication paths between objects to be established or changed dynamically.

AN OVERVIEW OF SINA

Basic concepts

Processes and objects are the basic building blocks of SINA. A process is an active entity executing a *process description*. A process description consists of a sequence of program statements that invoke operations on objects and control the flow of execution. Most of the control constructs are similar to those in Pascal.[19]

There are two kinds of objects in SINA, *passive* objects and *active* objects.* The object model in SINA is a non-uniform model, similar to those used in Argus,[10] Eden,[21] and CLU,[22] where the objects of the primitive types are treated differently from the application defined abstract data types. On the other hand, Smalltalk-80,[4] with a uniform object model, treats all objects in an identical fashion and all invocations of operations on objects are based on messages.

Passive objects are instances of primitive types that include integer, real, character and boolean. These can be introduced in any process description. Operations on these objects are performed as in any conventional programming language such as Pascal.

Active objects are application defined and are implemented using one or more processes. There are two kinds of active objects, *data objects* and *methods*. Data objects implement abstract data types; methods implement the operations on abstract data types, i.e. data objects contain methods. A data object is defined by a *type definition*. A type definition consists of two parts, an interface definition that declares all externally visible operations and a local definition that encapsulates the object's state and implementation. A local definition consists of three parts: a definition of all local objects, a process description of an initial method (used to initialize the object's state) and process descriptions of all local and interface methods. Any object can be nested within a data object, thus defining a hierarchical name-space. The visibility of names is based on Pascal-like block structure scope rules.

A method consists of a method-name, a process description and a (possibly empty) set of parameters. Methods are invoked similarly to procedures, except that a method invocation results in an invocation request message being sent to the method. Methods

* The use of the terms *passive object* and *active object* should not be confused with the use of these terms in Hydra.[23]

can process invocation requests either on a first-in-first-out basis or according to a priority scheme. The priority can be passed as an integer-valued parameter; a low value implies a high priority. Methods can use local passive objects to hold parameter values from the invoker of the method.

Active objects cannot be nested within a process description due to garbage collection problems that arise when a process terminates.†

Figure 1 shows an example of the structure of a type definition for an object that implements a stack of integers. The interface part declares two methods pop and push; the local part contains the definitions of these two methods. The local part also contains two passive objects: stackptr is an integer and stacklist is an array of size N of integers. The other details of this example will be clearer when we discuss the synchronization mechanism. For now this should give an intuitive notion of the structure of a type definition.

The general structure of a SINA program is described below using BNF notation.

```
⟨program⟩ ::= ⟨program heading⟩ ⟨constant definitions⟩ ⟨type definitions⟩ ⟨global
              object declarations⟩
              ⟨global process definitions⟩ ⟨main program⟩
```

```
constants N is 100;
type stacktype interface is
begin
  method push (integer as item) returns nil;
  method pop ( ) returns integer;
end;
type stacktype local is
begin
  objects integer as stacklist[N]; integer as stackptr;
  initial begin ^pop.hold( ); stackptr := 0; end;
  methods
    push: begin
            stacklist[stackptr] := item;
            stackptr := stackptr + 1;
            if (stackptr = N) then ^push.hold( );
            ^pop.accept( );
          end;
    pop:  begin
            stackptr := stackptr - 1;
            if (stackptr = 0) then ^pop.hold( );
            ^push.accept( );
            return stacklist[stackptr];
          end;
end;
```

Figure 1. Structure of a type definition module

† Without this restriction is is difficult to determine if it is safe to delete such a local active object when a process terminates because an active object can be deleted safely only when all of the local processes within that object have terminated.

The type definitions part of program defines all the active data types that are used by the program. The global object declarations section declares instances of these types. The following is an example of global object declaration where mystack and myqueue are declared as instances of stacktype and queuetype, respectively:

```
globals stacktype as mystack; queuetype as myqueue;
```

Global objects are visible only to the global processes and the main program. All global objects are created before the first statement of the main program is executed, and all active objects local to an active object are created before any operation on that object is executed.

The global process definitions section defines a (possibly empty) set of processes that are not encapsulated within any object. A process definition is quite similar to a procedure definition in Pascal; a list of parameters may be associated with a process. An instance of such a process is created by invoking that process in a fashion similar to calling a procedure. This results in the creation of a process that executes concurrently with its creator. Multiple instances of the same process may be created to execute concurrently. A SINA program may consist only of global objects and no global processes.

The main program section co-ordinates the program's execution by creating instances of the global processes, invoking operations on the global objects, or making a global object visible to another global object by giving it the pointer to that object.

Inter-object communication

Invoking the interface methods of an active object is the primary means by which another active object or process can communicate and change/access the state of that object. Invocations are based on messages using the *request–reply* model of communication. An invoked method can return an object of one of the primitive types to the invoker using the return statement. The nil object is returned in the case when a method does not explicitly return an object.

An object can communicate with another object by using either that object's name or a pointer to that object. Access by name is restricted to the scope of the name; so access to the global objects by name is available only to the main program and to any global process. An object can access itself by using the special name self. An example of access by name is the invocation

```
mystack.push(N)
```

This pushes the value N on an object named mystack, which is an instance of the stacktype described in Figure 1.

Object pointers provide access to active objects regardless of scope. Passive objects cannot be accessed by object pointers. Object pointers are typed capabilities that can be used dynamically to change or establish communication paths between objects. Since object pointers are typed, they allow access only to objects of the corresponding type. The following expression declares objptr to be an object pointer for stack stacktype:

```
@stacktype as objptr;
```

The next expression causes objptr to point to the instance mystack of stacktype:

```
objptr:=&mystack;
```

Such a pointer as &mystack is 'absolute' in the sense that it has a unique value.

Temporary pointers provide objects with temporary capabilities to access other objects. This facility is useful when an object wants to give another object temporary access to some of its local objects, bypassing the restrictions imposed by the object encapsulation mechanism. The following expression generates a temporary pointer temptr from the object pointer objptr:

```
temptr:=!objptr;
```

A temporary pointer can be viewed as an alias for an object pointer that exists until the temporary pointer is *destroyed*. Other temporary pointers can be generated from a temporary pointer. Thus temporary pointers form an *alias tree*.

The destruction of a temporary pointer can either be explicit or implicit. Any attempt to use a destroyed temporary pointer signals a run-time error. Explicit destruction is done by using the destroy function. A temporary pointer that is generated in the parameter list of an invocation is destroyed implicitly when the invocation completes. An important property of the destruction of a temporary pointer is that all the descendants of that pointer in the alias tree are destroyed as well.

Message processing and synchronization

Every active object has a mailbox for buffering invocation request messages. No assumption is made about the amount of buffer space available in the mailbox. Furthermore, every active object has an *object manager*, which is a system-defined object that schedules the processing of invocation request messages to the object it manages. Some system-defined operations can be invoked on an object manager for synchronization. The object manager for any object X is named as ^X.

In order to synchronize concurrent activities, an object's interface can be in one of two states: *hold* or *accept*. In the *hold* state, message processing at the object's interface is suspended indefinitely and all messages remain in its interface's mailbox. Messages for an object are scheduled for processing only if its interface is in the *accept* state. The interface of an object is put in the *hold* state when the hold operation is executed on the manager of that object. Similarly an interface is put in the *accept* state by executing the accept operation. The operation count on an object manager returns the count of messages in that object's mailbox.*

For example, for an object of type stacktype in Figure 1, the operation^ pop.hold() would inhibit processing of messages for the pop method. The push method is unaffected. The execution of ^pop.accept() resumes the processing of messages for pop.

By default all interfaces are initialized to the *accept* state. An object can cause *hold/accept* state changes for the interface of another object only if both these objects are encapsulated within the same type definition.

* In the current implementation these operations are supported only on the object managers of methods.

In the *accept* state the messages in an object's mailbox are processed one at a time. In this state the interface of an object alternates between two substates: *free* or *blocked*. In the *accept* state a message is removed for processing from the interface mailbox of an object only if the object's interface is *free*. An interface is *blocked* if a message to that interface is currently being processed by the object, otherwise it is said to be *free*. Processing of a message by an interface method will block the interfaces of that method and its encapsulating object. When an interface is blocked, all invocation request messages wait in the interface's mailbox. The next message is processed only when the interface is *free*.

For example, for an object stack of type stacktype, when an invocation message stack.pop() is being processed by stack and its method pop, no other messages are accepted for processing by any of the interface methods of stack until its interface is freed again.

Processing of a message by a method results in the creation a new process to execute the process description associated with that method. This process is referred to as the *triggered process*; it has its own stack and its own copies of all the local variables and parameters of the method. A triggered process terminates when it executes either its last statement or a return statement. When a triggered process terminates, all interfaces that were *blocked* by its message processing are *free* again.

With the rules presented so far there can only be one process executing within any object. This restriction is eliminated as described below. A triggered process can free all interfaces blocked by its message processing by executing the detach operation. After the execution of the detach operation the triggered process continues its execution and it is called a *detached process*. Processing of other messages can then resume at those interfaces after an execution of the detach operation. Thus new processes may be created within the object, executing concurrently with the detached processes.

A detached process carries the complete context of its invocation message to enable it to return the result to the invoker. This context also contains copies of the local variables and parameters; these variables are not shared with any other process. Any further execution of the detach operation by an already detached process does not cause any state change.

When an object is created, the initial method of the object is executed first before processing the invocation requests. An object can start processing requests at its interface only after the initial process has either terminated or executed the detach operation. Thus an initial process may continue executing concurrently with some other processes in its object.

A call to self within an interface method is allowed at run-time only if the process executing the method is detached. A local method can make a call to self only if the invoker of that local method can be allowed to make a call to self at that time. Notice the recursion in these conditions, since an interface method may call a local method, which may also call another local method.

Execution model for objects

The run-time execution model of an active object has four components: (1) a set of processes to implement the methods defined for that object, (2) a data area, shared by the processes executing methods, for storing the local passive objects, (3) pointers to local active objects and (4) an object manager process. The object manager process acts as the manager of the encapsulated object and all its interface methods. The run-time structure of an active object is illustrated in Figure 2.

An object manager responds to two types of messages: *request* and *control*. Request messages are sent by the invoker of an interface method. Control messages are sent by an object's initial process, or by any process executing a local or interface method of that object. Control messages are sent when a hold, accept, count or detach operation is invoked. Also, an implicit control message is sent to notify an object manager when the initial process or a triggered process completes execution. The object manager gives higher priority to the processing of control messages. While processing these request/control messages, the object manager is responsible for maintaining or releasing mutual exclusion to its object.

Each active object is assigned a unique identifier known as its *object identifier*. An important part of object management is the maintenance of the mailboxes of an object. These are handled through an *object mailbox* which is a mapping between the object's identifier and two mailboxes: a *request mailbox* and a *control mailbox*. This allows messages to be sent to an object on the basis of its object identifier. As is illustrated in Figure 3, one of the first things that an object manager does is to *connect* to an object mailbox.

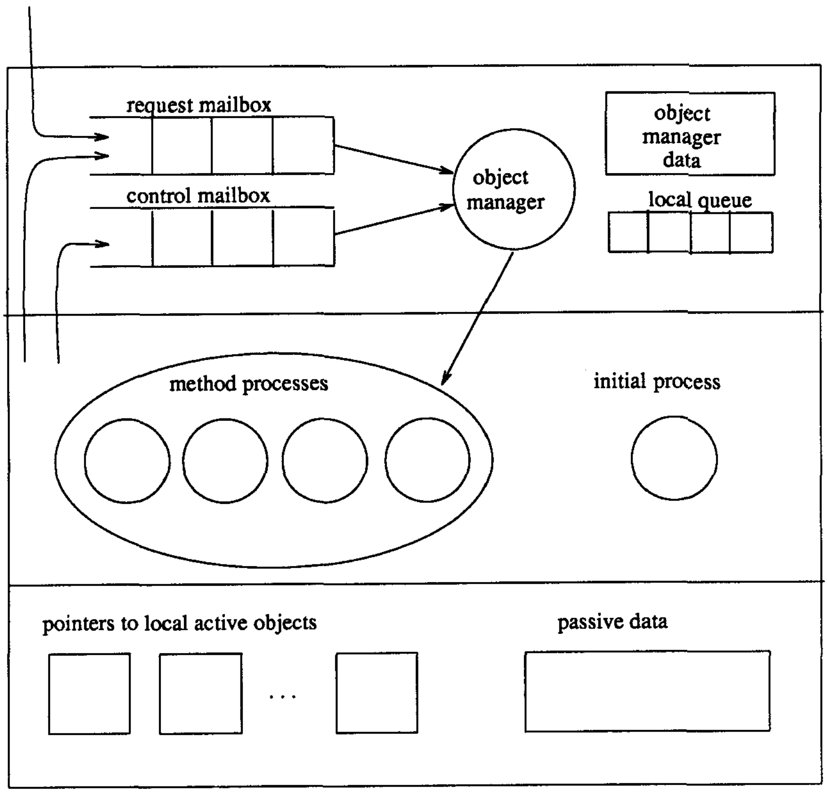

Figure 2. Run-time model for a SINA object

```
Connect to an object mailbox;
Create local active objects;
Create a process to execute the initial method;
Wait for the initial process to either complete or detach;

loop
    while there is a control message to be processed
        Service the control message;
    endwhile;

    while there is a request message
        Save the request in the local queue;
    endwhile;

    while there is a serviceable request in the local queue
        Remove a serviceable request from the local queue;
        Create a "triggered process" to service the request;

        loop
            Wait for a control message;
            if the control message indicates that the triggered process servicing the current request has completed or detached
            then exit from loop;
            else service the control message;
        endloop;
    endwhile;

    Wait for a request or control message;
endloop;
```

Figure 3. Control algorithm executed by an object manager

The object manager receives messages from these mailboxes and triggers processes to execute methods as appropriate. The processes executing methods have access to the object's local objects. Active objects local to an object are only logically nested in that object — not physically nested. Pointers to such local active objects are maintained by the nesting object. The object manager also maintains information about the states (*hold* or *accept*) of its interface methods, and the number of requests pending at its method interfaces.

A generalized description of the control algorithm executed by an object manager is given in Figure 3. Since messages from the mailboxes are removed on a first-in-first-out basis, a *local queue* of messages is maintained by the object manager to schedule the processing of request messages according to the current state of its interface methods. Requests are removed first-in-first-out for the interface methods in the *accept* state.

Local methods act as objects local to an active data object. Hence the run-time structure of a local method is a special case of an active object. A local method has an object manager, which can create a process to execute the process description of that method. The use of an object manager for a local method is necessary since the invocation of a local method should behave as though the call was made to an instance

```
program TokenRing;
constants N is 10;
type node interface is
begin
  method SendToken ( ) returns nil;
  method connect (@node as neighbor) returns nil;
end;

type node local is
begin
  objects
    @node as next;   /* next is a pointer for objects of node type */
    method token( ) returns nil;
  initial
    begin
      detach; ^token.hold( );
      while true do
        begin
           token( );
           /* execute critical section */
           @next.SendToken( ); /* send token to next node */
        end;
    end;
  methods
    connect:  begin next := neighbor; end;
    SendToken: begin ^token.accept( ); end;
    token:    begin ^token.hold( ); end;
end;
globals node as ring[N];
main program;
objects integer as i;
begin
  for i := 0 to N-1 do ring[i].connect(&ring[(i+1) mod N]);
  ring[0].SendToken( );
end.
```

Figure 4. Circulating token ring

of an active type, especially in the sense that the caller may be blocked at the interface of the local method. A call to detach in a local method affects only the interface of that method — not of the object wherein that method is defined.

EXAMPLES OF SINA PROGRAMS

This section presents some examples of SINA programs to illustrate the computation model of this language.

Circulating token ring

Figure 4 gives an example of a complete SINA program. Here we present a circulating token protocol for implementing mutual exclusion between a set of objects connected

```
program ProducerConsumer;
constants N is 10;

type buffer interface is
begin
     method append (integer as item) returns nil;
     method remove ( ) returns integer;
end;

type buffer local is
begin
     objects
        integer as itemcount, head, tail, buf[N];
     initial begin head := 0; tail := 0; itemcount := 0; ^remove.hold( ); end;
     methods
        remove:
        objects integer as val;
        begin
             itemcount := itemcount - 1; if itemcount = 0 then ^remove.hold( );
             val := buf[head]; head := (head + 1) mod N;
             ^append.accept( ); return val;
        end;
        append:
        begin
             itemcount := itemcount + 1; if itemcount = N then ^append.hold( );
             buf[tail] := item; tail:= (tail + 1) mod N;
             ^remove.accept( );
         end;
end;

globals buffer as queue;

process producer (integer as repetitions, delaytime);
objects integer as i;
begin
   delay(delaytime); /* suspend the producer for delaytime real-time seconds */
   for i := 1 to repetitions do begin queue.append(i); printf("deposited %d in the buffer", i); end;
end;

process consumer (integer as repetitions, delaytime);
objects integer as i, j;
begin
   delay(delaytime); /* suspend the consumer for delaytime real-time seconds */
  for j := 1 to repetitions do begin i := queue.remove( ); printf("received %d", i); end;
end;

main program;
objects integer as i;
begin
   producer(40, 5); producer(20, 2); consumer(50, 5); consumer(10, 10);
end.
```

Figure 5. SINA implementation of producer consumer synchronization using bounded buffer

in a ring. Each node is connected to the next node in the ring by sending it the appropriate object pointer. Then the token is sent to the first node in the ring by the main program. This token then circulates from one node to the next in the ring. A node executes the critical section in mutual exclusion when it has the token.

Communication using bounded buffer

This example, shown in Figure 5, gives a complete program implementing two producer and two consumer processes that synchronize using an object called queue, a bounded buffer of size 10. This buffer stores integer valued items. The bounded buffer is defined as an object with two interface methods append and remove. The producers call the interface method append to deposit a newly produced item into the buffer. Similarly, the consumers call the interface method remove to obtain an item from the buffer.

Initially the buffer is empty, so the initial process puts the remove interface in the *hold* state. Whenever an item is deposited by the append method, the remove interface is put in the *accept* state. The execution of the remove method once again puts this (i.e. remove) interface in the *hold* state if the count of items in the buffer becomes zero. Method remove puts the append interface in the *accept* state. Method append puts this (i.e. append) interface in the *hold* state if the buffer becomes full.

Alarm clock

In this example we use the feature that allows messages at an interface to be queued according to some specified priority. This example also illustrates use of local methods and the use of detach to permit multiple concurrent processes within an object. An alarm-clock object, as shown in Figure 6, has two interfaces tick and wakeme. The tick interface is called by some external clock object. The alarm-clock maintains the current time using an integer variable now. Every time tick is called, now is incremented by one. An external object calls the wakeme interface when it wants to be delayed for some number of ticks specified by the parameter delay. One alarm-clock object can be used concurrently by any number of clients.

A call to the wakeme interface creates a detached process in order to permit other requests to be processed by the alarm-clock. A detached process within wakeme calls the method test to check if its delay period has expired. The requests at the interface of test are ordered according to their checktime value. A request to test checks if now is either equal to or has exceeded its checktime. If so, then it returns true to the caller and leaves the interface of test in the *accept* state. Otherwise, it returns false and leaves test in the *hold* state. The interface of test is put in the *accept* state by the processing of a message by tick whenever tick is invoked. If an invocation of test by a process within wakeme returns true, the process terminates, returning nil to the object which called the wakeme interface. Otherwise, it once again sends an invocation request message to test to check its expiration time.

Factorial computation

This example, shown in Figure 7, illustrates the feature of invoking an interface method of an object from inside of that object by invoking operations on self. Notice that an object of type factorial does not contain any initialization method. It has one

```
type alarm interface is
begin
      method wakeme (integer as waittime) returns nil;
      method tick( ) returns nil;
end;

type alarm local is
begin
      objects integer as now;
              method test(priority as checktime) returns boolean;
      initial begin now := 0; end;
      methods
          test: begin
                  if now >= checktime then return true;
                  else begin ^test.hold( ); return false; end;
              end;
          tick: begin print("."); now := now + 1; ^test.accept( ); end;
          wakeme: objects integer as alarmsetting;
                begin
                    alarmsetting := now + waittime;
                    detach;
                    while not test(alarmsetting) do   ;
                end;
end;
```

Figure 6. Alarm clock

```
type factorial interface is
begin
      method eval (integer as i) returns integer;
end;

type factorial local is
begin
      methods
          eval: objects integer as j;
              begin
                if i = 0 then return 1;
                else begin detach; j := i * self.eval(i-1); return j; end;
              end;
end;
```

Figure 7. An example of invocation on self

interface method named eval, which returns the factorial of its integer valued parameter. This method computes factorial recursively, each recursive call results in a new triggered process.

Client–server communication

This example illustrates the utility of the temporary pointers when a client object wishes to give a server object temporary access to some of its local objects during the execution of an invocation. Once the server completes processing the request, it loses access through such temporary pointers.

This example shows only one part of a large SINA program where a person's database is maintained in an object called PersonalData, as shown in Figure 8. This object maintains information about a person's finances. It also maintains a pointer, called MyAccountant, to an object of type Accountant (not shown here) that provides some accounting and tax preparation service to this object. This object also contains an object called mytax of type TaxForm, and an object myincome of type Income. We have omitted the definition of these active objects since their logical characteristics should be clear.

When the interface method FileYourTaxes is invoked on an object of type PersonalData, it requests the operation PrepareTax to be invoked on the Accountant object pointed to

```
type PersonalData interface is
begin
   method PutName (char as name[10]) returns nil;
   method AddIncome (integer as dollars) returns nil;
   method GetIncome ( ) returns integer;
   method FileYourTaxes ( ) returns integer;
   method YourAccountant (@Accountant as who) returns nil;
end;

type PersonalData local is
begin
   objects char as myname[10];
        IncomeData as myincome;
        TaxForm as mytax;
        @Accountant as MyAccountant; /* MyAccountant is a pointer for objects of Accountant type */
   methods
     PutName: objects integer as i; begin for i:=0 to 9 do myname[i]:=name[i]; end;
     AddIncome: begin myincome.AddIncome(dollars); end;
     GetIncome: begin return myincome.GetIncome( ); end;
     YourAccountant: begin MyAccountant := who; end;
     FileYourTaxes: objects integer as i;
             begin
                @MyAccountant.PrepareTax(!&myincome, !&mytax);
                i := mytax.GetTax( ); printf("my tax is %d", i); return i;
             end;
end;
```

Figure 8. Object representing an individual's finances

```
type Database interface is
begin
   method read ( ) returns nil; /* this method prints the current value of the database */
   method write (integer as x) returns nil;
end;

type Database local is
begin
    objects
       integer as data;
       RWsynch as coordinator;
    methods
       read: begin detach; coordinator.readstart( ); printf("%d", data); coordinator.readfinish( ); end;
       write: begin detach; coordinator.writestart( ); data := x; coordinator.writefinish( ); end;
end;
```

Figure 9. Shared database object

by the pointer MyAccountant. It gives temporary pointers for its local objects myincome and mytax to the server as parameters of the PrepareTax operation. Thus the accountant object has access to these two objects only during the processing of the request message. Once the accountant responds to the invocation message, these temporary pointers become invalid.

Reader–writer synchronization

This example shows how a resource and the mechanism to synchronize access to it can be encapsulated within the same object. This is possible owing to the support for intra-object concurrency in SINA. This contrasts with the monitor-based solutions where typically a resource and the monitor to synchronize access to it have to be separate modules.

A shared database is concurrently accessed by some readers and writers. The definition of the shared database object is shown in Figure 9; it contains an integer named data and an object named coordinator, an instance of the type RWsynch, that is used for synchronizing readers and writers.

In a shared database object a process executing the read method first executes detach. It then executes a read operation on data. Similarly the write method on the database is executed by first executing detach, followed by the execution of a write operation on data. Each read or write operation is preceded by the execution of a readstart or writestart operation, respectively, on the coordinator object. Similarly, each read or write operation on data is followed by the execution of a readfinish or writefinish operation, respectively, on the coordinator object. Because the processes executing the read or write operations on data are detached, other invocation messages to the database object may be processed concurrently.

The type definition for RWsynch is presented in Figure 10. The reader–writer synchronization policy is implemented by this object. Notice that in this example readers can block the writers indefinitely; we have shown implementations of other policies in another paper.[24]

```
type RWsynch interface is
begin
  method readstart( ) returns nil;
  method readfinish( ) returns nil;
  method writestart( ) returns nil;
  method writefinish( ) returns nil;
end;

type RWsynch local is
begin
   objects integer as readcount;
   initial begin readcount := 0; end;
   methods
      readstart: begin readcount:=readcount+1; if (readcount=1) then ^writestart.hold( );  end;
      readfinish: begin readcount := readcount-1; if (readcount=0) then ^writestart.accept( ); end;
      writestart: begin ^readstart.hold( ); ^writestart.hold( ); end;
      writefinish: begin ^writestart.accept( ); ^readstart.accept( ); end;

end;
```

Figure 10. Reader–writer synchronization object

IMPLEMENTATION MODEL

SINA is currently implemented on Sun-3 and Apollo workstations using the Concurrent C programming language,[25] which is an extension of the C programming language[26] with support for concurrent programming. Using Concurrent C as the implementation base, a simple message-passing facility is constructed for supporting the SINA run-time execution environment. A SINA program is translated to a Concurrent C program, which is executed with the SINA run-time library.

The implementation model has two parts: run-time support and program translation. The run-time implementation model has three major components: process management and synchronization, a message-passing facility and object management. This model is illustrated in Figure 11. This section describes these three components of the run-time support.

Process management and synchronization

The process management scheme of the Concurrent C System is based on ‘lightweight processes’, i.e. concurrency is supported by the language, not the underlying operating system. Concurrent C provides the following features for process management and synchronization. We use these features to describe our implementation model. The following mechanism is provided by Concurrent C for creating a process:

```
createProcess(processname(parameter1, . . ., parameter N))
```

This creates the named process and starts its execution with the given parameters. This is analogous to a function call with the exception that createProcess can return

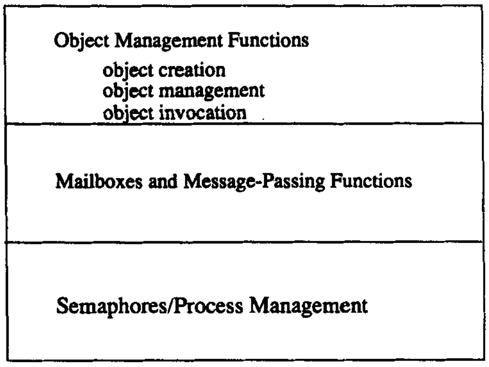
Object Management Functions
object creation
object management
object invocation
Mailboxes and Message-Passing Functions
Semaphores/Process Management

Figure 11. SINA run-time environment

before the process has completed execution. Process synchronization in Concurrent C is based on semaphores.[27] There is a function for creating a semaphore; for example, the following will create a semaphore whose initial value is initval:

```
sem := createSemaphore(initval)
```

Then sem can be used in calls to wait and signal. The function wait_or allows a process to wait on either of two semaphores, i.e. wait_or(sem1,sem2) will block until either sem1 or sem2 is non-zero.

Mailboxes and message-passing

Mailboxes are implemented using the semaphore facility of Concurrent C. Each mailbox is identified by a *mailbox identifier*. Functions are provided to send messages to or receive messages from a mailbox. These operations may block a sender if the mailbox is full or a receiver if there are no messages in the mailbox. It is also possible for a process to wait to receive a message from any of two specified mailboxes; the receiver process blocks until either of the two specified mailboxes is non-empty. Also, functions are provided for the allocation and de-allocation of mailboxes.

Using the mailbox facility just described, object mailboxes are implemented. As previously stated, the two components of an object mailbox are a request mailbox and a control mailbox. This facility allows objects to communicate with one another on the basis of object identifiers only. Functions are provided by the object mailbox facility to check if a request or a control message is currently present in an object's mailbox. Also, functions are provided to send or receive request and control messages to an object or to wait until the request mailbox or control mailbox of an object is non-empty.

Object management

Object management involves the following two major functions: object creation and method invocation.

Object creation

The underlying model of object creation is dynamic; all active objects are created dynamically using the run-time functions described below. For each type there is a function in the run-time environment to create an object of that type and return the object identifier of the newly created instance. For each active object, the corresponding creation function (for that type) must be called before the object is used. All active objects local to an active object must be created before the initial process for that object is executed.

If AType is an active type, then the function ATypeCreate creates a new instance of that type and returns the object identifier of the new instance. This function has the following responsibilities: allocation of an object identifier, creation of the object manager process, and synchronization with the object manager's initialization. Synchronization with the object manager amounts to waiting for the object manager to do two things: create all the active objects local to the object, and wait for the initialization process to detach or finish execution. An outline of the creation function for a type AType is as follows:

```
sync := createSemaphore(0);
newid := allocObjectId(); /* create a new object object identifier */
createProcess(ATypeObjectManager(newid,sync) ); /* create an object manager
process for an instance of AType */
wait(sync);
return newid;
```

Thus when the object manager of the new instance has completed its initialization, it must signal the semaphore sync. Notice the recursion in this description: for an active object to be created, it must wait for its local active objects to be created, which must do likewise.

Method invocation

The invocation of an object's local and interface methods is supported by a *call function* in the run-time environment. For each method there exists a call function that has the following responsibilities: packing the parameters of the invocation into a request message, sending a request message to the object, waiting for a reply, and returning the result of the invocation, if appropriate. For example, consider the stack example in Figure 1. The method push would have the call function stacktypePushCall which takes two parameters: the object identifier of the object to be called, and the parameter to the push operation. In this case, stacktypePushCall would have no return value. So the statements

```
stacktype as stk;
stk.push(num);
i := stk.pop();
```

would be translated as

```
stk := createStacktype();
stacktypePushCall(stk,num);
i := stacktypePopCall(stk);
```

The second statement would result in the call function for the push operation doing the following:

1. Pack the parameter num into a request message.
2. Send the message to the object stk.
3. Wait for a response from stk.
4. Return the *return value* from the invocation (not appropriate in this case).

Now we present a more detailed description of the invocation of an object's interface methods. First we consider the point of view of the caller, i.e. the one invoking the method. The following steps must be taken:

1. Build a request message:
   - (a) identify the method to be called
   - (b) generate temporary pointers as parameters (if any)
   - (c) pack the parameters to the method
   - (d) specify a *return mailbox* for returning the response.
2. Send the request message to the *object mailbox* of the callee object.
3. Wait until the response message is received in the *return mailbox*.
4. Get the return value of the method.
5. Destroy temporary pointers, if any, that were generated as parameters.

The following complementary actions are taken by the object manager of the callee object:

1. Get a request message from the object mailbox.
2. Put the request message into the object manager's local queue.
3. Select a request message from the local queue.
4. Create a 'triggered process' to execute the appropriate method to service the request.
   When the triggered process completes serving the request
   - (a) it first sends a message to the specified *return mailbox*.
   - (b) it then sends a control message to its object manager indicating its completion.

RELATED WORK

In the current implementation we were primarily concerned with the concurrency, synchronization and inter-object communication features of SINA. The mechanisms of hold, accept and detach are novel and provide flexibility for supporting concurrency within an object. These mechanisms for synchronizing concurrent processes are an integral part of the object-oriented computation model of SINA.

In this paper we have presented a number of examples illustrating how these mechanisms are used. These examples have been taken from SINA programs that have been fully tested using our implementation. In a separate report[28] we have shown that these constructs are as powerful as semaphores and monitors by showing implementations of these two well-known mechanisms using SINA. The SINA support for intra-object concurrency is particularly useful in programming hierarchically nested resources. Also, in another paper[24] we have demonstrated the usefulness of SINA constructs in programming different synchronization policies for a number of well-known problems in communication, scheduling and hierarchical resource management.

In contrast to SINA, Smalltalk-80 supports concurrent programming by introducing two distinctly different conceptual entities: *objects* and *processes*. They represent two different levels of abstractions: processes are *actors* and objects are *servers*.[3] A process is created by sending a *fork* message to a block context. Also, a number of primitives are provided for process scheduling. Semaphores are used for process synchronization. Concurrent Smalltalk, which is upward compatible with Smalltalk-80, attempts to provide one single conceptual level for concurrent programming that consists of objects only. It supports asynchronous method calls and introduces *CBox* objects to synchronize with the responses of asynchronous invocations and avoid use of semaphores. However, it introduces two distinct categories of objects: *atomic objects* and *non-atomic objects*. The atomic objects act as serializers; messages to such objects are processed sequentially. The non-atomic objects are compatible with objects in Smalltalk-80; processing of invocation messages to such objects is concurrent.

In contrast, all active objects in SINA are treated alike. One can build systems consisting only of active objects without any global processes. Global processes in SINA are introduced only for the convenience of abstraction and programming simplicity. SINA does not provide any special kinds of objects, such as semaphores, serializers or atomic objects, for synchronization. If needed, such objects can be built using the primitives provided by SINA.

There are a number of similarities between the object models of Argus, CLU and SINA. All these languages have a non-uniform view of object management. CLU does not support any concurrency. Argus and SINA both support intra-object concurrency. Nevertheless, the mechanisms used in these two languages for intra-object synchronization are distinctively different. For synchronization, Argus relies solely on the atomicity of concurrent actions. Concurrent atomic actions in Argus are synchronized using locking protocols,[29, 30] which are transparent to the programmer. Such locking protocols are suitable in transaction-oriented database applications but not in solving inter-process communication problems in operating systems. In comparison, the synchronization mechanisms provided by SINA are more basic, and the synchronization of actions using these mechanisms is completely left to the programmer.

Eden, which is a distributed operating system, also supports object-oriented computing based on the notions of abstract data types, as in Argus and SINA. In Eden, the synchronization within an object is based on the monitor construct of Concurrent Euclid.[31] Thus the synchronization of nested objects in Eden has the same limitations as those associated with nested monitor structures.

In the Actor model[32] concurrent processes are supported within a shared resource module using the *serializer* construct. A serializer is an extension to the monitor concept;[33] it controls access to a shared resource by concurrent users. There are some similarities between a serializer and an object manager in SINA. The most important distinction between a serializer and a SINA object manager is that, in SINA, an object manager is itself treated as an object whose behaviour is altered dynamically by invoking operations on it.

EVALUATION

We would like to briefly mentioned our experiences with the system and the lessons we learned from this project. During the course of this work we learned that it was helpful to start with a problem-oriented goal. Specifically that we started with a set of SINA programs that we wanted to run. Our design and implementation was guided by the features that we found useful through these examples. For example, we dropped nested methods since we did not come up with any examples where they were particularly helpful. Also we added object pointers (and temporary pointers) since we had very concrete examples of where the implementation without pointers was inadequate.

Our initial design did not include the concept of object pointers. Without object pointers global objects can only be accessed by invoking operations on them from within global processes. This appeared to be too restrictive. We felt that it should be possible to build systems that consist of objects only, without any global processes, and allow objects to interact with one another. Through our attempts to program certain resource management problems the value of object pointers became clear.

Temporary pointers allow an object to give another object a temporary capability. This is a somewhat primitive facility as compared to the elaborate capability-based mechanisms included in the Hydra design. In the current implementation, the holder of a temporary pointer to an object is given complete access to that object during the lifetime of that pointer. In future we would like to include constructs to restrict access through such temporary pointers to a subset of the object's interface functions.

Some syntactic changes to the language were conceived during the implementation phase of the project. The earlier definition of SINA had a single syntactic unit describe the interface and the local parts of a type definition. The introduction of object pointers allowed two type definitions to refer to each other through pointers. To translate such mutual references we either had to resort to two-pass syntax analysis or to some kind of foward declaration construct. We realized that separating the interface definition part of a type definition from the local definition and allowing the interface part to appear anywhere before the local declaration would serve the purpose of forward declaration with more elegance.

It was during the implementation phase that we realized the problems of having local active objects in methods. These problems, as described earlier in the paper, were related to garbage collection of such objects when the process executing such a method terminates.

We would like to make a few comments on the development path adopted in this project and compare it with another implementation effort[2] for SINA using Smalltalk-80 that we undertook later. In the system described here a SINA program is translated to Concurrent C and then compiled. This system is clearly much faster in terms of the execution time; however, we found that it was much easier to modify the Smalltalk-based implementation to experiment with new language features.

There is one important lesson that we learned from the comparative evaluation of these two implementations. The implementation described here does not allow dynamic binding of an object pointer to different types of objects. The invocation messages are prepared, and then interpreted, using the compile-time generated procedures that are strongly tied to the type definition of the target object whose method is being invoked. This requires object pointers to be typed; thus disallowing the flexibility of dynamically accessing different types of objects using the same object pointer. We found this to be a major limitation of our implementation.

Finally, we want to note that the creation of all active objects in SINA is based on the dynamic object creation facility in the run-time environment; however, dynamic object creation is currently not available to the programmer. It would be a simple and obvious extension to our implementation to provide this.

In this paper we have omitted several other object-oriented features of SINA. The general idea behind the data abstraction model of SINA is that, starting from a simple object-based model, one can implement various forms of abstractions without committing oneself to a fixed number of techniques such as inheritance, delegations and relations. We tend to agree with the views expressed by some other researchers[14, 3] that class inheritance, though an important and crucial concept in object-oriented programming, is secondary to the idea of encapsulation. Therefore no language constructs have been adopted in SINA for specifying inheritance or delegation. Instead, it provides a mechanism called *message predicates*[2] that supports single and multiple inheritance, delegation and relations. These features have been implemented under a separate project, and the results of that work are described in Reference 2.

ACKNOWLEDGEMENTS

We would like to thank the two anonymous referees for their very helpful comments and suggestions for restructuring and revising this paper. Thanks are also due to Dick Wolfe for his comments and help in revising this paper.

REFERENCES

1. Mehmet Aksit and Anand Tripathi, 'An overview of the SINA language', *Technical Report*, Oce'-Netherlands, Venlo, Holland, November 1986.
2. Mehmet Aksit and Anand Tripathi, 'Data abstraction mechanisms in SINA/ST', *Proceedings of the ACM OOPSLA'88 Conference*, September 1988.
3. Grady Booch, 'Object-oriented development', *IEEE Trans. Software Engineering*, **SE-12**, (2), 211–221 (1986).
4. A. Goldberg and David Robson, *Smalltalk-80, The Language Design and Implementation*, Addison-Wesley Publishing Company, 1983.
5. B. J. Cox, *Object-Oriented Programming*, Addison-Wesley, Reading, Massachusetts, 1986.
6. B. Stroustrup, 'An overview of C++', *SIGPLAN Notices*, **21**, (10), 7–18 (1986).
7. C. Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian and Carrie Wilpolt, 'An introduction to Trelis/Owl', *OOPSLA'86*, September 1986, pp. 9–16.
8. D. G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Misinter, Mark Stefik and Frank Zdybel, 'CommonLoops, merging Lisp and object-oriented programming', *OOPSLA'86*, 1986.
9. A. Snyder, 'Commonobjects: an overview', *SIGPLAN Notices*, **21**, (10), 19–28 (1986).
10. M. Tokoro and Y. Ishikawa, 'Orient84/K: a language with multiple paradigms in the object framework', *19th Hawaii Conference on System Sciences*, January 1986, pp. 198–207.
11. Akinori Yonezawa, Etsuya Shibayama, Toshihiro Takada and Yasuaki Honda, 'Modelling and programming in an object-oriented concurrent language ABCL/1', in Akinori Yonezawa and Mario Tokoro (eds), *Object-Oriented Concurrent Programming*, MIT Press, Cambridge, MA, 1987, pp. 91–128.
12. Y. Yokote and M. Tokoro, 'Concurrent programming in Concurrent Smalltalk', in Akinori Yonezawa and Mario Tokoro (eds), *Object-Oriented Concurrent Programming*, MIT Press, Cambridge, MA, 1987, pp. 129–158.
13. Y. Yokote and M. Tokoro, 'Experience and evolution of Concurrent Smalltalk', *OOPSLA'87*, October 1987, pp. 406–415.
14. O. M. Nierstrasz, 'Active objects in Hybrid', *OOPSLA'87*, October 1987, pp. 243–253.
15. P. America, 'POOL-T: a parallel object-oriented language', in Akinori Yonezawa and Mario Tokoro (eds), *Object-Oriented Concurrent Programming*, MIT Press, Cambridge, Massachusetts, 1987, pp. 199–220.
16. Andrew Lister, 'The problem of nested monitor calls', *Operating Systems Review*, ACM SIGOPS, July 1977, pp. 5–7.
17. Bruce K. Haddon, 'Nested monitor calls', *Operating Systems Review*, October 1977, pp. 18–23.
18. F. B. Schneider and A. J. Bernstein, 'Scheduling in Concurrent Pascal', *Operating Systems Review*, ACM SIGOPS, April 1978, pp. 15–20.
19. Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report*, Springer-Verlag, 1985.

20. Barbara Liskov, 'Overview of the Argus language and system', MIT, Laboratory for Computer Science, Cambridge, MA 02139, February 1984.
21. G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, 'The Eden system: a technical review', *IEEE Trans. Software Engineering,* **SE-11**, 43–59 (1985).
22. Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler and Alan Snyder, *CLU Reference Manual*, Springer-Verlag, 1981.
23. William Wulf, Roy Levin and Samuel Harbison, *HYDRA/C.mmp: An Experimental Computer System*, McGraw-Hill Book Company, 1981.
24. Anand Tripathi and Mehmet Aksit, 'Communication, scheduling, and resource management in SINA, *Journal of Object-Oriented Programming,* **1**, (4), 24–41 (1988).
25. Choi-Kwok Chan and Anand Tripathi, 'Design and implementation of the Concurrent C system', Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, October 1987.
26. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
27. E. W. Dijkstra, 'Cooperating sequential processes', in F. Genuys (ed.), *Programming Languages*, Academic Press, 1968, pp. 43–112.
28. Anand R. Tripathi and Mehmet Aksit, 'Concurrent programming and synchronization in an object-oriented model of computing', Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, January 1988.
29. K. P. Eswaran, J. N. Gray, R. A. Lorie and I. A. Traiger, 'The notion of consistency and predicate locks in database systems', *Communications of the ACM,* **19**, 624–633 (1976).
30. Eliot Moss, 'Nested transactions: an approach to reliable distributed computing', *MIT/LCS/TR-260*, Massachusetts Institute of Technology, Laboratory of Computer Science, Cambridge, MA 02139, April 1981.
31. R. C. Holt, *Concurrent Euclid, The UNIX System, and Tunis*, Addison-Wesley Publishing Company, 1983.
32. Russell Atkinson and Carl Hewitt, 'Synchronization in actor systems', *ACM Proceedings of Principles of Programming Language*, January 1977, pp. 267–280.
33. C. A. R. Hoare, 'Monitors: an operating system structuring concept', *Communications of the ACM,* **17**, 549–557 (1974).