Communication, Scheduling, and Resource Management in SINA

Anand Tripathi*
Mehmet Aksit†

Introduction

In this article we present the object-oriented constructs for communication and concurrent programming in the SINA programming language. The object encapsulation mechanism of SINA allows implementations of hierarchically structured resource management systems and alleviates some of the problems that arise in using nested monitor structures [13,28,31]. SINA supports capability-based communication between objects using object pointers. This permits building systems in which the communication topology can change dynamically. Using these constructs we present object-oriented solutions to various problems in interprocess communication, scheduling, and resource management. A number of examples of such problems have been implemented and tested using our SINA system on the SUN 3 workstations.

In contrast to monitors [16], which are conventionally viewed as passive entities, the SINA objects are active entities that may contain multiple concurrent processes. In the past, several new mechanisms and models modifying the monitor construct [13,28,31,32] have been proposed to remedy the problems that arise in the nested monitor structures. However, we share the views expressed by Parmas [29] that it is more desirable to provide a set of basic mechanisms and language constructs that can be conveniently used to build "tailored monitors" with some desired synchronization and scheduling requirements rather than try to fit a particular monitor construct to some general requirements. The main contribution of the SINA design is in this direction to define such a set of basic mechanisms and language constructs. In another work [34] we have shown how semaphores and monitors can be implemented using the SINA constructs, thus demonstrating that the SINA constructs are as powerful as these two well-known mechanisms.

The object pointer facility in SINA provides capability-based communication and is useful for dynamically establishing or changing the interconnection of objects. This facility also allows one to establish "temporary" binding between a client and a server object. In SINA, a client while requesting some service from a server object can give to the server "temporary" capabilities for some of its local objects. Such capabilities are valid only during the performance of the requested service by the server. It is also possible for the client to restrict the operations permitted to be performed by the server on the client's local objects by channeling the access by the server through another object.

This article is organized as follows. The next section describes the background and related work in the area of concurrent programming and synchronization. The third section presents the basic concepts in our computation model and the mechanisms to support concurrency and synchronization within SINA objects. The remaining three sections contain various examples to demonstrate the capabilities of the proposed mechanisms to solve some well-known synchronization problems. Examples in three different categories—communication, scheduling, and resource management—are presented in these three sections. These examples also illustrate how one can implement hierarchically structured resource management systems in SINA, which are typically difficult to implement using nested monitors.

Modules, Objects, and Concurrency

The problems of concurrent programming and synchronization have been extensively studied during the past two decades. The various mechanisms and constructs proposed and used to solve these problems are based on either shared memory or message-passing. In shared memory systems, elegant solutions to the various synchronization problems can be constructed using semaphores [11], conditional critical regions [18], path expressions [8], modules [35], monitors [14,16], and event counters [30]. Various different models of concurrent programming and synchronization based on message-passing have been proposed and used in the past in Communicating Sequential Processes (CSP) [17], Ada [20], actors [4], Distributed Processes (DP)[15], Argus [26], Synchronizing Resources (SR)[3], and managers [21].
The concept of monitor has been used in Concurrent Pascal [14], MESA [24], and Modula [35] as a mechanism for synchronizing access to shared resources by concurrent processes. The monitor construct is useful in shared memory systems because a monitor is a passive entity that encapsulates some shared data and provides interface procedures to access that data. A process can gain access to the data structures of a monitor only by executing its interface procedures. In this sense a monitor defines an abstract data type. Inside a monitor, at any time at most one process can be active. Processes synchronize by executing the monitor procedures and, within these procedures, by executing wait or signal operations on some condition variables. In a monitor, a process executing the wait operation on some condition is blocked in the queue corresponding to that condition; a signal operation on a condition causes exactly one such waiting process to be resumed and the signaling process itself is suspended. This has some implications, as discussed below, on defining nested monitor structures.

For certain problems the monitor construct imposes restrictions and limitations on hierarchical system structuring, for example, in case of a shared database system, the monitor to synchronize the readers and the writers can not encapsulate the database itself; it has to be implemented separately from the database. This restriction arises from the fact that there can be at most one active process inside a monitor and thus it is not possible to have multiple active readers inside a monitor that encapsulates the database itself.

The problem described above is a form of the general problem (or the non-problem [29]) of nested monitor calls[13,28]. When one defines a monitor which nests another monitor, a deadlock situation can possibly arise when a call to the nested monitor is made from within the nesting monitor and the procedure inside that nested monitor executes a wait operation. This can leave the outer monitor in the blocked state and no other process can enter the monitor any longer. Nesting of monitors also has the effect of limiting parallelism and can hinder implementation of certain scheduling policies. One example of such a problem [31] arises in Concurrent Pascal [14] when one attempts to build a disk scheduler monitor that encapsulates the disk I/O system. Such a system structure prohibits implementation of a disk scheduling algorithm that may require reordering of the pending requests according to some scheduling priority.

Process synchronization using messages is based on either synchronous (unbuffered messages) or asynchronous (buffered message) message passing. In case of synchronous communication, the sender (receiver) process trying to send (receive) a message to (from) another process is blocked until that process is ready to communicate. When both processes are ready to communicate, the communication takes place by copying the message from the address space of the sender into the address space of the receiver. This synchronization model is used in the Communicating Sequential Processes (CSP). Another form of synchronous communication requires that the sender process be blocked until it receives a response message from the receiver. Such a model is useful in systems based on the request-reply paradigm of computing. This model has been used in Distributed Processes (DP), Ada, Argus, and Synchronizing Resources (SR). In case of asynchronous communication, the sender process is not blocked if the receiver is not ready to communicate. The sender is free to perform other computations after the message has been accepted by the system kernel. Similarly, a receiver is not blocked if no message is available at the time of execution of the receive operation.

The data abstraction concepts have been supported by Simula [7], CLU [27], Alphard [36], Smalltalk [12], and several other recent programming languages for sequential computing. The request-reply model unifies the concepts of data abstraction and message-based interprocess communication for distributed and concurrent programming. A module is no longer treated as a passive entity whose interface procedures are directly executed by its client processes. A module is now viewed as an active entity, To execute an interface procedure of a module, a client process sends an invocation request message to that module. One or more processes are created in the module to handle the request messages. On receiving a request message one such process executes the requested interface procedure on behalf of the caller and sends all the updated values of the var parameters back to the caller. The caller is blocked until it receives this reply message. This mode of communication is termed as remote procedure call (RPC) [6].

The concept of encapsulating a shared resource as an active module which reacts to request messages from its clients forms the basis of object-oriented concurrent and distributed programming. In the object model of computing [23] objects are instances of abstract data type modules. The policies for the synchronization of concurrent operations on this module are implemented by the concrete representation of the object. Several concurrent and distributed programming languages (Argus [26], Path Pascal [9], ABCL [37,38], Act1 [25]), and distributed operating systems (Eden [2], ISIS [3], Nexus [33]) are based on this kind of approach. The design of SINA is also based on this view of object-oriented computing.

For synchronizing concurrent access to the internal state of the object these designs have included a range of different mechanisms. For example, in Argus and Nexus atomic actions are used as the primary mechanism for synchronization within an object. Objects in Eden are constructed using Concurrent Euclid [19] which provides the monitor construct for synchronization. Distributed Processes (DP) and Synchronizing Resources (SR) use boolean expressions as guards for synchronization. A guard determines whether a request for some interface procedure can proceed with its execution. In Path Pascal objects are passive entities and path-expressions [8] is the mechanism used for synchronizing access to an object. In the actor model a mechanism called serializer [4] was introduced for synchronization. All messages to an actor (which represents the shared resource object) are relayed through its serializer. The serializer mechanism can be viewed as a derivative of the monitor concept. In comparison to monitors, the actor model supports multiple processes concurrently executing the actor operations.

Object-Oriented Programming Concepts in SINA

The SINA language provides some primitive object types which include integer, real, character, and boolean. It also has a princi-
tive data structuring construct array for defining a list of elements of the same type. Objects and processes are the basic building blocks of a SINA system. Type and method definitions are the mechanisms for supporting encapsulation of objects to define more complex, application-defined, objects in the system.

The object model in SINA is a non-uniform model, similar to those used in Argus, Eden, and CLU. Smalltalk [12] with a uniform object model treats all objects in an identical fashion and the invocations of operations on objects are based on messages. On the other hand, in a non-uniform model, objects of the primitive types are treated differently from the application-defined objects. In the non-uniform object models of SINA, Argus, CLU, and Eden, objects of the primitive types are implemented as in any conventional programming language by allocating storage for them in the address space allocated to the process or module in which they are declared. The operations on these objects are performed as in any conventional programming language such as Pascal [22]. In comparison to this, for all application-defined objects, the operation invocations are message-based. These basic concepts in SINA's computation model are described below.

**Objects:** Informally, objects belong to two categories: data objects and methods. Data objects are instances of either the system-defined primitive types or some application-defined abstract data type modules. Methods are modules that provide functional abstraction similar to functions in Pascal [22]. However, methods are quite different from Pascal functions in the following way. A function in Pascal is a sequence of code that is executed by the calling process. On the other hand, a method in SINA is an active entity which is invoked based on the remote procedure call model. An object may be nested within other objects, thus defining hierarchical name spaces. The visibility of names is based on Pascal-like block structure scope rules.

**Processes:** A process is an active entity executing a process description. A process description consists of a sequence of program statements that invoke operations on the objects visible according to the scope rules and control the flow of execution of the program statements. Most of these constructs in SINA are quite identical to those in Pascal. A process can also execute the delay statement which causes it to be suspended for the specified number of seconds.

**Methods:** Methods provide functional abstraction in our system. A method is a module with a method name, a process description, and possibly some other data objects nested within it. A process invokes a method in a fashion identical to calling a function in Pascal. This results in sending an invocation message to the object representing that method. Some of the nested objects of the method are used for holding parameter values from the sender of the invocation message. The requests are queued at the interface of the method either in the first-in-first-out order or according to their priority order if a priority level is associated with the requests to that method. It is possible to use one of the integer valued parameters of a method as a priority level for the requests to that method. A lower value for this parameter would imply a higher priority.

**Abstract Data Types:** An abstract data type defines a module consisting of some objects (which we refer to as the local objects of that module). Some of the local methods can be made visible to the outside objects by declaring them in the interface definition part of the module. Only those methods whose names appear in the interface part of a module can be invoked by the external objects. The state of an abstract data object can be changed or accessed by an external object only by invoking such interface methods.

**Messages and Operation Invocations:** The invocation of a method is based on the remote procedure call model of synchronous communication. The messages to an object are queued at its interface. No assumption is made about the amount of buffer space available for queuing messages. Invocation of a method results in sending an invocation message to the interface queue of that object. An object or process interacts with another object by invoking its interface methods. This requires that in the invocation statement the method name be qualified with the name of the object. In this case, the request message is sent to the interface queue of that object. The manager of this object then schedules this message for processing by sending it to the queue of the invoked interface method.

For example, a user of some resource would acquire the resource from the resource manager RM by executing the operation $RM.acquire\ (myid)$, where $myid$ is the id of the client. This results in sending a request message to the object $RM$. At some later time $RM$ would accept the message and send it to $myid$. The final receiver of the message, i.e., $myid$ in this case, executes the requested operation and returns the result of the invocation.

**Message Processing:** When a message is received by a method, a process is created within the method according to the process trigger rules, described in the section detailing message processing and concurrency, to execute the process description associated with the method. This process has its own stack and its own copies of all the local variables and parameter objects of that method. A process terminates when it executes either its last statement or a return statement. The return statement is used for returning an object, which is the result of the invocation, to the sender of the message. Execution of the return statement with no arguments, or termination of the triggered process without executing a return statement, causes a system-defined constant object $nil$ to be returned to the invoker.

A SINA program consists of the definitions of: (1) some abstract data types; (2) global objects which are instances of such abstract data types; (3) global processes (that are not part of any object) to carry out computations involving the global objects (it is possible to build systems which consist of objects only because objects are active entities); and (4) the main program which schedules the global processes and connects the global objects by giving them pointers to other global objects to establish communication paths between them. In the current implementation the number of objects is statically fixed, but their communication structure can change and evolve dynamically.

**Types Definitions in SINA**

A type definition describes a generic module for a class of objects which have some common structure and attributes. A type definition consists of two parts: interface definition and
local definition. The interface part declares all the interface methods. The local definition consists of three parts: definition of all local objects, an initialization process, and process descriptions of all local and interface methods. Local data objects can be introduced within a method or any process description. Listing 1 shows an example of the structure of the type definition modules. The full description of this module, which implements a bounded buffer of integers, will be presented in a later section.

A method in an object can send a request message to any of the interface methods of that object by invoking that interface operation on an object called self. At run-time this is allowed only if the dynamic chain of invocations that resulted in the creation of the calling process is in the detached state (as described in the section detailing message processing and concurrency) at the object’s interface. This rule is introduced to avoid any possibility of deadlocks.

Object Pointers

Object pointers are capabilities which play an important role in defining the communication topology among objects in a SINA program. A process can invoke operations on an object if it has a pointer for that object. Pointer objects in SINA are typed in the sense that a pointer can point only to the objects of a specific type. In SINA, pointers can only be obtained for the abstract data objects.

In SINA the pointer for an object X can be obtained by evaluating the expression &X; for a pointer P, the object pointed to by it is given by the expression @P. A pointer object P for type T is declared in the object declaration part of a program or module as

    objects @T as P;

Object pointers can be passed as parameters to methods. If an object possesses a pointer to some object, then it can pass that pointer to some third object. An object can also build a “temporary” pointer for another object if it has a pointer for that object or if that object is visible according to the scope rules. For example, if some pointer p is pointing to some object X, then &p would generate a temporary pointer for object X. A temporary pointer q can be destroyed using the statement destroy(q); after a pointer has been destroyed, it becomes invalid for the remaining life-time of the system. Any subsequent attempt to access through this pointer would cause runtime error.

Example 1

    q := &p; /* this will generate a temporary pointer aliased to p */
    ....
    ....
    destroy(q); /* after this statement q becomes an invalid pointer */

A temporary pointer can be created and passed as a parameter while invoking a method. This is shown in Example 1. The callee object is permitted to access through such a temporary pointer (passed to it as a parameter) only during the life-time of the triggered process created for that invocation. Once the call returns,

Listing 1—Bounded Buffer

    constants N is 100;
    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;
the pointer possessed by the callee becomes invalid. In Example 2, the invocation of method \( m \) of some object \( X \) would give a temporary pointer for object \( A \) to object \( X \) and its method \( m \).

**Example 2:**

\[
p := \&A; /* p is pointer for the objects of the type of A */ X.m(p); /* Ip will generate a temporary pointer, aliased to p during the invocation */
\]

This pointer is valid only for the duration of the invocation and is automatically destroyed when method \( m \) returns. On the other hand, the invocation shown in Example 3 would give a permanent pointer for \( A \) to \( X \).

**Example 3:**

\[
p := \&A; \;
X.m(p);
\]

In Example 4, \( p \) is assigned a temporary pointer to \( A \). This pointer is passed as a parameter to object \( X \). In this case object \( X \) can retain this pointer for \( A \) and can continue to access \( A \) even after method \( m \) has returned. This retained pointer would be a valid pointer as long as no one has destroyed it. In this example, when \( \text{destroy}(p) \) is executed, the pointer value retained by \( X \) is rendered invalid.

**Example 4:**

\[
p := \&A; \;
X.m(p); \;
\ldots \;
\ldots \;
\text{destroy}(p);
\]

**Synchronization in SINA Objects**

A system-defined object manager is associated with every object. The object manager implements scheduling policies by selecting messages from the interface queue for processing by its object. For an object \( X \), its manager is referred to as \( \exists X \). In order to synchronize concurrent activities, the object's interface can be in one of two states: \( \text{hold} \) and \( \text{accept} \). In the \( \text{hold} \) state message processing at the object's interface is suspended indefinitely and all messages remain in the interface's queue.

Messages are accepted by an object for processing only if its interface is in the \( \text{accept} \) state. The interface of an object is put in the \( \text{hold} \) state when \( \text{hold} \) operation is executed on the manager of that object. An interface is put in \( \text{accept} \) state by executing the \( \text{accept} \) operation on its object manager. For example, \( \exists X.\text{hold}() \) will cause the interface of \( X \) to be put in the \( \text{hold} \) state, and \( \exists X.\text{accept}() \) will put the interface of \( X \) in the \( \text{accept} \) state. \( \exists X.\text{count}() \) returns the count of messages in the queue of object \( X \)'s interface. An object manager always processes all the pending request messages in own queue first, and then only if schedules processing of messages for its object. Among all the acceptable messages in the object's interface queue, the next message to be processed is selected on the first-come-first-serve basis.

For example, for an object \( \text{buf} \) of type \( \text{buffer} \) in Listing 1, the operation \( \exists \text{buf}.\text{hold}() \) would inhibit processing of messages from the interface queue of \( \text{buf} \). This means that no new invocation messages would reach the two interface methods \( \text{append} \) and \( \text{remove} \). On the other hand, if the interface of \( \text{buf} \) is in the \( \text{accept} \) state then the execution of \( \exists \text{buf}.\text{hold}() \) would inhibit processing of messages by only the method \( \text{append} \). The interface method \( \text{remove} \) would continue to receive and process messages from its interface queue.

By default all interfaces are initialized to the \( \text{accept} \) state. An object can send invocation messages to the \( \text{hold/accept} \) interfaces of the manager of some other object only if: (1) that object is visible according to the scope rules; and (2) both objects are nested within the same type definition module. Thus a process cannot cause \( \text{hold/accept} \) state changes for the interfaces of the objects outside its type definition module.

**Message Processing and Concurrency**

In the \( \text{accept} \) state the messages at an object's interface are processed one at a time. In this state the interface of an object alternates between two sub-states: \( \text{free} \) and \( \text{blocked} \). In the \( \text{accept} \) state a message is removed for processing from the interface queue of an object only if the object's interface is \( \text{free} \). An interface is \( \text{blocked} \) if a message to that interface is currently being processed by the object, otherwise it is said to be \( \text{free} \). When an interface is \( \text{block} \), all invocation messages wait in a queue at the interface. The next message is processed only when the interface is \( \text{free} \). A message to a nested object must pass through the interfaces of its encapsulating objects and will block their interfaces. For an object \( \text{buf} \) of type \( \text{buffer} \), when an invocation message \( \exists \text{buf}.\text{remove}() \) is being processed by \( \text{buf} \) and its method \( \text{remove} \), no other messages are accepted for processing at the interfaces of these two objects until their interfaces are freed again.

Processing of a message by an object results in the creation of a new process in the object and \( \text{triggering} \) it to execute the process description associated with that object. When this triggered process terminates, all interfaces which were \( \text{blocked} \) by this message processing are \( \text{free} \) again. In the above example a process is created inside the method \( \text{remove} \) to execute the process description of \( \text{remove} \), and interfaces of both \( \text{buf} \) and \( \text{remove} \) are blocked. This means that, 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 \( \text{detach} \) statement. After the execution of the \( \text{detach} \) statement the triggered process continues its execution and it is called a detached process. A detached process carries the complete context of the invocation message to enable it to return the result to the invoker. This context also contains copies of the local variables and the interface parameters; these variables are not shared with any other process. Because all interfaces blocked by the processing of a message are freed when the corresponding triggered process detaches, processing of other messages can resume at those interfaces after the execution of the \( \text{detach} \) statement. This can result in the creation of new processes, within the object, executing concurrently with the detached process. Any further execution of the \( \text{detach} \) statement by an already detached process does not cause any state change.
When an object is created, a process within that object is triggered to execute the initialization procedure specified in the type definition. An object can start processing messages at its interface only after the initialization process has either terminated or executed the detach statement. If the initialization process executes the detach statement, it is possible that the initialization process continues executing concurrently with some other triggered or detached processes.

Process Communication

This section presents three examples of interprocess communication and coordination. The first example implements a bounded buffer of integers. The second example basically illustrates the utility of object pointers in establishing dynamic communication structures. The third example shows how one can implement asynchronous operation invocations in SINA. This example once again shows the utility of the object pointers and the pointer parameters to methods.

Bounded Buffer

This example is shown in Listing 1—a buffer object with a capacity to store up to 100 integers. This object has two interface methods append and remove. This object is used by concurrently executing producer and consumer objects. The producers call the append interface to deposit an integer, and the consumers call the remove interface to obtain an integer from the buffer. The interface remove is initialized to the hold state. An append operation causes the remove interface to go into the accept state. Also, it causes the interface append to go into the hold state if the buffer becomes full. An execution of the process description of remove causes the append interface to go into the accept state. Also, it causes the remove interface to go into the hold state if the buffer becomes empty. Notice that at any time at most one process is executing inside the buffer object.

Tax Returns: An Example of Client-Server Communication Using Object Pointers

This example illustrates how client-server relationships can be dynamically established using object pointers. It also shows how using temporary pointers an object can give to another object (say, some kind of server) access to some of its local objects during the invocation of a service request. Finally, this example also shows how one can restrict such an access only to a subset of operations defined for such a local object.

In this example, we consider the client-server relationship between an individual and his tax accountant. The individual approaches the tax accountant for preparing his taxes. For this purpose he gives to the tax-accountant read-only access to its income database and complete access to its tax-form object to be filed or modified by the tax accountant. It is required that the tax accountant ceases to have access to these objects once the transaction of preparing the tax-return is over.

Listing 2—Income Database Object

```pascal
type IncomeData interface is
begin
  method AddIncome (integer as dollars) returns nil;
  method PutIncome (integer as dollars) returns nil;
  method GetIncome () returns integer;
end;

type IncomeData local is
begin
  objects integer as income;

  methods
    AddIncome: begin income := income + dollars;end;
    PutIncome: begin income := dollars; end;
    GetIncome: begin return income; end;
  end;
```

An individual's financial database contains an object of the type IncomeData as shown in Listing 2. This object maintains the income data of the individual. Using the interface methods AddIncome, PutIncome, and GetIncome this data can be accessed or changed.

Listing 3 shows the object type ViewIncome which is used to provide the read-only access to some IncomeData object. Such an object has stored inside it a pointer, called viewpoint, to an object of type IncomeData to which it provides the read-only access. This pointer is set from outside using the method connect which requires as a parameter a pointer to some IncomeData object. An invocation of the interface method GetIncome results in the invocation of the GetIncome method on the object currently pointed to by this internally stored pointer.

Listing 3—An Object to Provide the Read-Only View of an Income Database Object

```pascal
type ViewIncome interface is
begin
  method connect (@IncomeData is incomepntr) returns nil;
  method GetIncome () returns integer;
end;

type ViewIncome local is
begin
  objects @IncomeData as viewpoint;
  initial begin viewpoint := null; end;

  methods
    connect: begin viewpoint := incomepntr; end;
    GetIncome: begin return @viewpoint.GetIncome(); end;
  end;
```
Listing 4—Tax Form Object

```plaintext
type TaxForm interface is
  begin
    method PutTax (integer as amount) returns nil;
    method GetTax () returns integer;
  end;
end;

type TaxForm local is
  begin
    objects integer as taxes;
    initial begin taxes := 0; end;
    methods
      PutTax: begin taxes := amount; end;
      GetTax: begin return taxes; end;
  end;
```

The object type `TaxForm` is shown in Listing 4. The tax data is stored in it using the interface method `PutTax`, and this data can be read by invoking `GetTax`. The object type `Accountant` is shown in Listing 4. It has no local state variables. A client of this object calls the interface method `PrepareTax` to get his tax returns prepared. This method has two pointer parameters. These parameters are used by the invoker to give pointers to two of its local objects of type `ViewIncome` and `TaxForm`. Through this local object of type `ViewIncome` the client gives to the accountant read-only access to its private `IncomeData` object.

Listing 5 shows the `Accountant` type of object. It has one interface method called `PrepareTax` which is invoked by its clients objects. The invoker of this method gives two pointers as the parameters of the invocation. One pointer points to a `ViewIncome` object and the other points to a `TaxForm` object. Through the first pointer, the accountant reads the income database of the client. It then computes the tax and fills the `TaxForm` object of the client by invoking its interface method `PutTax`.

Listing 6 shows a client object of type `PersonalData`. This object contains inside it one IncomeData object, one ViewIncome object, one TaxForm object, and one pointer to an accountant object. This pointer is set from outside to point to an Accountant object that would prepare the tax-returns for this client. This binding can be changed dynamically using the interface method `YourAccountant`, if our client is not happy with his current accountant.

Income and any other relevant data are stored in an object of type `PersonalData` using interface methods `AddIncome`, `GetIncome`, and `PutName`. The initial method initializes the ReadMyIncome object to contain pointer to `myincome` object. When the interface method `FileYourTaxes` is invoked, the client of this type invokes the `PrepareTax` method of the accountant object currently pointed to by the pointer `MyAccountant`. It passes as parameters to this method temporary pointers to `ReadMyIncome` and `mytax`.

Listing 7 shows the main program part which creates two global objects `JohnSmith` and `U$ave` of the types `PersonalData` and `Accountant`, respectively. `JohnSmith` is given a pointer to `U$ave` by the main program.

Asynchronous Operation Invocations in SINA

The operation invocation model in SINA, and also in most of the other object-oriented programming languages, supports only procedure-call-like synchronous invocations. In this example we show how operations can be invoked asynchronously in SINA. This example illustrates the utility of the object pointers in implementing asynchronous invocations of operations on objects.

For this example we have an object type called `multiplier`, as shown in Listing 8, that supports one operation called `mul` for multiplying two integer numbers. This operation returns the product of the two integers given by its parameters. This figure also shows another object type called `RequestHandler` that supports asynchronous multiplication operations. It has two interface methods: `MultiRequest` and `GetResult`.

To make an asynchronous invocation of the `mul` operation on some object of type `multiplier`, the invoker would call the method `MultiRequest` of an object of the `RequestHandler` type. This invocation request has three parameters. The first gives a pointer to an object of type `multiplier` on which the `mul` operation is to be invoked, and the other two are integer-valued parameters specifying the two numbers to be multiplied.

The method `MultiRequest` copies all parameters into the local state variables on the `RequestHandler` object and then unblocks the initial process which is waiting at the method `gate`. Multi-

Listing 5—Accountant Object to Prepare Tax Returns

```plaintext
type Accountant interface is
  begin
    method PrepareTax (@ViewIncome as income; @TaxForm as tax) returns nil;
  end;
end;

type Accountant local is
  begin
    methods
      PrepareTax: objects integer as i;
      begin i := @income.GetIncome(); i := (i*25)/100; /* tax rate is 25% of income*/
      @tax.PutTax(i);
    end;
```

30 JOOP November/December 1986
Listing 6—Client Object to Provide Personal Data

type PersonalData 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;
ViewIncome as ReadMyIncome;
@Accountant as MyAccountant;
initial begin ReadMyIncome.connect(&myincome); end;
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(&ReadMyIncome,&mytax); i:=mytax.GetTax(); printf("my tax is %d", i);
return i; end;
end;

Listing 7—Main Program Defining Global Objects and Their Interconnection

globals Accountant as USave; PersonalData as JohnSmith;
main program;
objects integer as i;
begin
JohnSmith.YourAccountant (&USave);
JohnSmith.AddIncome (2000);
i := JohnSmith.FileYourTaxes();
printf("John Smith's tax is %d", i);
end.

Request then returns leaving the interface of both MultiRequest and GetRequest in the blocked state. At some later time the process requesting asynchronous multiplication invokes the GetRequest method to get the result of multiplication. The initial process, after completing the invocation of the multi operation on the multiplier object specified by the requester, and after storing the result locally, puts the interface of GetRequest in the accept state. The stored result is returned to the requester by the method GetRequest, which also puts the MultiRequest interface back into the accept state.

An object of type RequestHandler handles only one asynchronous call request at any time. In Listing 9 we implement an object called AsyncMult that encapsulates N number of such objects. Thus this new object can handle up to N such asynchronous call requests.

Scheduling Problems

This section presents some examples of scheduling problems where the order of execution of requests is required to be re-arranged to satisfy certain scheduling policies. In general, the language constructs to implement server objects and to handle request messages within such an object should allow the server to process the request messages, and to respond to them, in an order that may be different from the order of arrival of the request messages. In this respect the model used in SINA has a distinct advantage over the rendezvous mechanism in Ada [20].

In Ada, when a rendezvous between a client and a server task occurs, the server can not rendezvous with another task until it has completed processing of the current request and has responded to it. The alarm clock example given in the follow-
Listing 8—Asynchronous Invocation Handler

```plaintext
costants N is 10;
type multiplier interface is 
begin method mult (integer as i,j) returns integer; end;
type request local is 
begin method mult: begin return (i*)); end; end;
type request Handler interface is 
begin method MultRequest (@multiplier as compute; integer as i,j) returns nil;
    method GetResult () returns integer;
end;
type request Handler local is 
begin 
    objects integer as x,y,result; @multiplier as server;
    method gate () returns nil;
    initial begin Tgate:=hold(); detach;while true do begin gate(); result :=server.mult(x,y); TGetResult.accept(); end; end;
    methods 
        gate: begin Tgate:=hold(); end;
        MultRequest: begin TMultiRequest:=hold(); x:=i; y:=j; server := compute; Tgate:=accept(); TGetResult:=hold(); end;
        GetResult: begin TMultiRequest:=accept(); return result; end;
end;
```

This section illustrates the merit of the object-oriented constructs of SINA. The implementation of such an alarm clock object in Ada is somewhat tedious, but definitely not impossible.

**Alarm Clock**

An alarm clock object, as shown in Listing 10, has two interfaces `tick` and `wake`. 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 `wake` 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 `wake` interface results in a detached process in order to permit other requests to be processed by the alarm clock. A detached process within `wake` calls 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. Other-

Listing 9—Multiple Asynchronous Invocation Handler

```plaintext
type AsyncMult interface is 

    type AsyncMult interface is 
    begin method asyncmult (@multiplier as compute; integer as k,j) returns integer;
        method Getresult (integer as id) returns integer;
    end;
type AsyncMult local is 
begin 
    objects RequestHandler as server[N]; boolean as free[N]; integer as freecount;
    initial objects integer as i; begin for i := 0 to N−1 do free[i] :=true; end;
    methods 
        asyncmult: objects integer as i;
        begin 
            freecount := freecount−1; if freecount=0 then Tasyncmult:=hold();
            i:= 0; while (not free[i]) and (i<N) do i := i+1;
            free[i] := false; detach; server[i].MultiRequest(compute,k,j); return i;
        end;
        Getresult: objects integer as i;
        begin i := server[i].GetResult(); free[i] := true; freecount := freecount+1; Tasyncmult:=accept(); return i; end; end;
```
Listing 10—Alarm Clock

```pascal
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 printf("\n"); now := now + 1; test.accept(); end;
wakeme: objects integer as alarmsetting;
  begin alarmsetting := now + waittime; detach; while not test(alarmsetting) do; end;
end;
```

wise, it returns \textit{false} and leaves \textit{test} in the \textit{hold} state. The interface of \textit{test} is put in the \textit{accept} state by the execution of \textit{tick}. If a request to \textit{test} by a process within \textit{wakeme} returns \textit{true}, the process terminates returning \textit{nil} to the object which called the \textit{wakeme} interface. Otherwise, it again sends a request message to \textit{test} to check its expiration time.

\textbf{Reader-Writer Synchronization: A Fair Solution}

A set of readers and writers access an object called \textit{SharedData} by sending messages to its interface \textit{read} and \textit{write}. The problem is to synchronize readers and writers \cite{10} such that the readers and the writers execute in mutual exclusion and similarly any two writers execute in mutual exclusion. The readers are allowed to read concurrently. In another work \cite{34}, we presented a SINA program to solve this problem where the waiting writers are given priority over the new readers. The scheduling policy in that example can cause the readers to be blocked forever. In this example we implement a policy that is a bit more fair. The implementation of this policy is reflected in the design of the object type \textit{RWSynch}. The policy is stated below:

In the absence of any waiting writers, all readers are allowed to proceed and read concurrently. If a write request arrives, then no new reader is allowed to proceed and when all current readers have terminated the first waiting writer in the interface queue is allowed to perform write operations. Every time a writer finishes, the next operation request is selected strictly on the first-come-first-serve basis.

The definition of shared data object is given in Listing 11. This object encapsulates two other objects: \textit{database} which stores the data shared by the readers and the writers, and \textit{coordinator}, an object of type \textit{RWSynch}, which is used for synchronizing read and write requests. The processes triggered within \textit{read} and \textit{write} interfaces detach themselves immediately after creation to allow

Listing 11—Shared Data Object and Read/Writer Synchronization

```pascal

```

begin
  begin detach; coordinator.readstart(); p := a; q := b; r := c; id := readerid;
  printf("%d,%d,%d",p,q,r); coordinator.readfinish(); end;
write: begin detach; coordinator.writestart(); a := x; b := y; C := z; coordinator.writefinish(); end;
end;
```

November/December 1988  JOOP  33
Listing 12—Reader/Writer Synchronization

```lisp
(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();
                    if ↑writestart.count() > 0 then ↑readstart.hold(); end;
        readfinish: begin readcount:=readcount-1; if (readcount=0) then ↑writestart.accept(); end;
        writestart: begin ↑readstart.hold(); ↑writestart.hold(); end;
        writefinish: begin ↑readstart.accept(); ↑writestart.accept(); end;
end;
```

Other read and write requests to be processed concurrently by SharedData. The solution is required to enforce writer-writer exclusion and reader-writer exclusion.

Listing 12 shows the object type RWSynch. An object of RWSynch type has four interface methods: readstart, readfinish, writestart, writefinish. The concurrent processes within SharedData synchronize with each other using these interfaces. The process associated with the read interface of SharedData object calls the readstart interface of the coordinator, then performs the desired read, and finally calls the readfinish interface. The process description of write interface is analogous.

A readstart request increases the count of currently reading readers. If a reader in readstart finds readcount equal to 1, meaning that it is the first reader in the current batch of concurrent readers, it then puts the writestart interface in the hold state. Thus no writer would be able to complete invocation of this method while a reader is active. The readstart method also checks if there is any waiting writer; if so, then it puts the readstart interface in the hold state to prevent new readers from starting their read operations. A writestart request puts both the writestart and readstart interfaces in the hold state. The invocation of writefinish by a writer puts both writestart and readstart interfaces in the accept state.

Every time a writer finishes, the next operation (a read or a write) is selected in the first-come-first-served order. If the next request is for readstart and there is some waiting writer after it, then that request message will be processed and no other reader would be allowed to proceed until a waiting writer gets a chance to write. Thus this solution still gives some priority to the writers, but it does not allow the writers to block the readers indefinitely.

Listing 13—Resource Manager Object

```lisp
(type RManager interface is
begin method acquire (priority as need) returns nil;
    method release () returns integer;
end;
(type ResourceManager local is
begin
    objects integer as usage;
    method done() returns nil;
    initial begin ↑done.hold(); detach; while true do begin done(); delay(1); usage := usage+1; end; end;
    methods
        done: begin end;
        acquire: begin ↑acquire.hold(); usage := 0; ↑done.accept(); end;
        release: begin ↑done.hold(); ↑acquire.accept(); return usage; end;
end;
```
Listing 14—Dynamic Allocation of a Pool of Identical Resources

```
|type RManager interface is |
|begin|
|  method request() returns @ResourceType; /* returns pointer to a resource instance, null pointer means no resource was allocated*/|
|  method release(@ResourceType as respnt) returns integer; /* returns total usage time*/|
|end;|
|type RManager local is |
|begin|
|  objects boolean as free[N]; usage[N]; ResourceType as ResourcePool[N]; @ResourceType as allocation[N];|
|  initial objects integer as i;|
|  begin avail:=N; for i:=0 to N-1 do begin free[i]:=true; allocation[i]:=null; end;|
|  detach; while true do begin delay(1); for i:=0 to N-1 do if not free[i] then usage[i]:=usage[i]+1; end;|
|  methods|
|  request: objects integer as i; @ResourceType as P;|
|  begin|
|    i:= 0; while (not free[i]) and (i<N) do i:=i+1;|
|    if (i=N) then return null;|
|    else begin avail := avail-1; P:= !&ResourcePool[i]; allocation[i]:=P; free[i]:=false; usage[i]:=0; return P; end;|
|  end;|
|  release: objects integer as i;|
|  begin|
|    if respnt<>null then|
|      begin i:= 0; while (i<N) and (respnt<>allocation[i]) do i:=i+1;|
|      if i<N then begin destroy(respnt); avail:=avail+1; free[i]:=true; return usage[i]; end;|
|      end;|
|      end;|
|end;
```

Resource Management Problems

In this section we present two examples to implement resource schedulers. In the first example the resource scheduler manages one resource; in the second example it deals with the allocation of resources from a pool of identical, reusable, resources.

Resource Allocator

This example, given in Listing 13, shows a resource controller object that controls a single resource. A client requests the resource by invoking the interface method acquire and indicating the duration of the job through this method’s parameter. This parameter is used as the priority level to schedule the requests with the “shortest job next” policy. The description of this resource manager object is given in Listing 13. Requests on the acquire interface are accepted only if the resource is currently available. While a resource is in use, internally the initial process of the resource manager object keeps the count of the amount of time the resource has been in use by the current user. This usage count is returned to the user when it invokes release. The purpose of the internal method done is to allow the initial process to start counting only when the resource is in use.

Allocation of Reusable Resources Using Object Pointers

This example illustrates the use of object pointers in allocating resources from a pool of resources. This example, shown in Listing 14, implements a resource management object that manages a pool of identical resources of some abstract type called ResourceType. In [34] we presented another solution to this problem without using object pointers.

Our implementation satisfies the following requirements as stated in [32]. (1) At any time only one process may be accessing a given instance from the pool. (2) A client process should not be able to find out the identity of the particular instance of the resource allocated to it from the pool. (3) A process currently accessing an instance from the resource pool should not be able to block another process from accessing a different instance from the pool which has already been allocated to that process. We add one more requirement, i.e., (4) once a client returns a resource back to the pool, it ceases to have access to that resource by any means.

A user requests an instance from the pool by executing the interface method acquire. This returns a (temporary) pointer to a resource instance in the pool. The user process then accesses the instance by invoking operations on that instance through this pointer. Finally the user process releases the instance allo-
Cated to it by executing the interface method release and specifying the resource pointer given to it. The resource manager identifies the resource allocated corresponding to that pointer, marks it free, and then it returns the amount of time for which this resource was used by this particular user. The initial process keeps count of the time of usage for each allocated resource instance.

Conclusion

In this article we have presented the basic computation model of SINA and its constructs for communication and synchronization. In implementing systems with nested resource structures, these constructs alleviate the difficulties that arise in using monitor-based solutions. The object pointer facility in SINA allows one to build dynamic communication structures in the system. The temporary pointer facility is particularly useful in establishing temporary client-server interactions between objects. The synchronization constructs in SINA permit implementations of objects with different policies for synchronization. We have illustrated the versatility of these constructs using a number of examples; a large number of such examples have been programmed and tested using our implementation of SINA on the SUN 3 workstation.

There are several other aspects of the SINA language that we have not presented in this paper. These are related to the data abstraction and type inheritance mechanisms in this language [1].

Acknowledgments

The SINA language was implemented by Eric Berge on the SUN 3 workstation as his Master's degree project at the University of Minnesota. This implementation is based on Concurrent C, a concurrent programming language based on C, which was implemented by Choi-Kwock Chan as his Master's degree project at the University of Minnesota.

References

33. Anand Tripathi, (November 1987) An Overview of the Neos Distributed Operating System, Department of Computer Science, University of Minnesota, Minneapolis, MN.
34. Anand R. Tripathi and Mehmet Akst, (January 1988) Concurrent Programming and Synchronization in an Object-Oriented Model of
Computing, Department of Computer Science, University of Minnesota, Minneapolis, MN.


