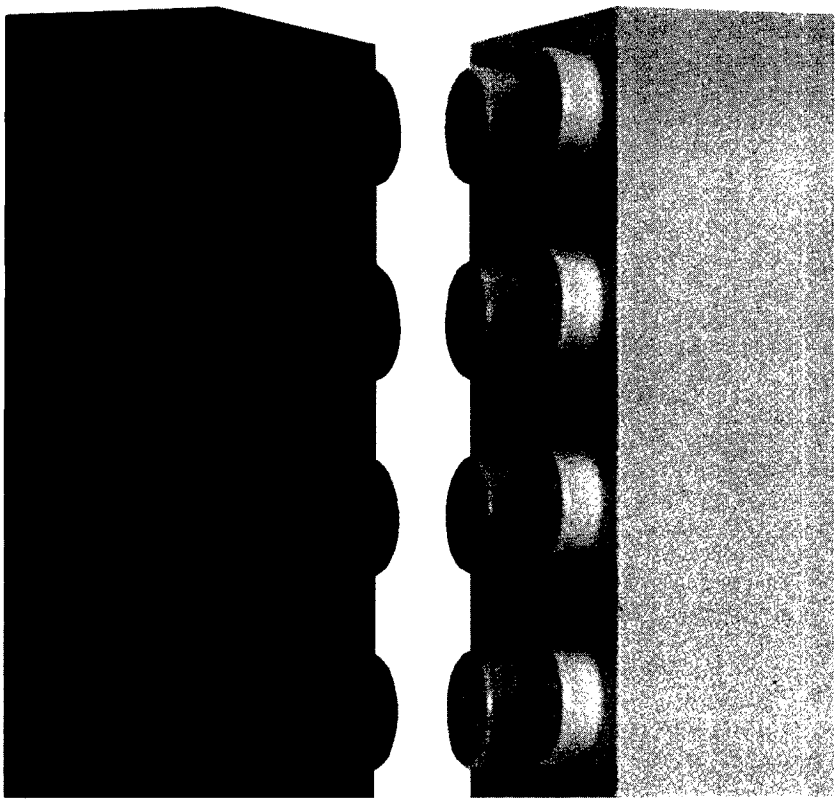

CONCURRENT OBJECT-ORIENTED PROGRAMMING

Three significant trends have underscored the central role of concurrency in computing. First, there is increased use of interacting processes by individual users, for example, application programs running on X windows. Second, workstation networks have become a cost-effective mechanism for resource sharing and distributed problem solving. For example, loosely coupled problems, such as finding all the factors of large prime numbers, have been solved by utilizing ideal cycles on networks of hundreds of workstations. A loosely coupled problem is one which can be easily partitioned into many smaller subproblems so that interactions between the subproblems is quite limited. Finally, multiprocessor tech-

nology has advanced to the point of providing supercomputing power at a fraction of the traditional cost.

At the same time, software engineering considerations such as the need for data abstraction to promote program modularity underlie the rapid acceptance of object-oriented programming methodology. By separating the specification of what is done (the *abstraction*) from how it is done (the *implementation*), the concept of objects provides modularity necessary for programming in the large. It turns out that concurrency is a natural consequence of the concept of objects. In fact Simula, the first object-oriented language, simulated a simple form of concurrency using coroutines on conventional architectures. Current development of concurrent object-oriented programming (COOP) is providing a solid software foundation for concurrent computing on multiprocessors. Future generation computing systems are likely to be based on the foundations being developed by this emerging software technology.

The goal of this article is to discuss the foundations and methodology of COOP. *Concurrency* refers to the potentially parallel execution of parts of a computation. In a concurrent computation, the components of a program may be executed sequentially, or they may be executed in parallel. Concurrency provides us with the flexibility to interleave the execution of components of a program on a single processor, or to distribute it among several processors. Concurrency abstracts away some of the details in an execution, allowing us to concentrate on conceptual issues without having to be concerned with a particular order of execution which may result from the quirks of a given system.

Objects can be defined as entities which encapsulate data and operations into a single computational unit. Object models differ in how the internal behavior of objects is specified. Further, models of concurrent computation based on objects must specify how the objects interact, and different design concerns have led to different models of communication between objects. Object-oriented programming builds on the concepts of objects by supporting patterns of reuse and classification, for example, through the use of inheritance which allows all instances of a particular class to share the same method.

In the following section, we outline some common patterns of concurrent problem solving. These patterns can be easily expressed in terms of the rich variety of structures provided by COOP. In particular, we discuss the actor model as a framework for concurrent systems[1] and some concepts which are useful in building actor systems. We will then describe some other models of objects and their relation to the actor model along with novel techniques for supporting reusability and modularity in concurrent object-oriented programming. The last section briefly outlines some major on-going projects in COOP.

It is important to note that the actor languages give special emphasis to developing flexible program structures which simplify reasoning about programs. By reasoning we do not narrowly restrict ourselves to the problem of program verification—an important program of research whose direct practical utility has yet to be established. Rather our interest is in the ability to understand the properties of software because of clarity in the structure of the code. Such an understanding may be gained by reasoning either informally or formally about programs. The ease with which we can carry out such reasoning is aided by two factors: by modularity in code which is the result of the ability to separate design concerns, and by the ability to abstract program structures which occur repeatedly. In particular, because of their flexible structure, actor languages are particularly well-suited to rapid prototyping applications.

[1]The term *Actor* was introduced by Carl Hewitt at MIT in the early 1970s to describe the concept of reasoning agents. It has been refined over the years into a model of concurrency. It should be noted that our use of the term bears no relation to the *language* Actor—the latter being a commercial product introduced in the late 1980s.

Patterns of Concurrent Problem Solving

Three common patterns of parallelism in problems have been found in practice (For example, see [8, 13]). First, *pipeline concurrency* involves the enumeration of potential solutions and the concurrent testing of these solutions as they are enumerated. Second, divide and conquer concurrency involves the concurrent elaboration of different subproblems and the joining of (some or all) of their solutions in order to obtain a solution to the overall problem. In divide and conquer concurrency, there is no interaction between the procedures solving the subproblems. A third pattern can be characterized as *cooperative problem-solving*. Cooperative problem-solving involves a dynamic complex interconnection network. As each object carries out its own computational process, it may communicate with other objects, for example, to share the intermediate results it has computed. An example of this kind of a system is a simulation where the physical objects are represented by logical (computational) objects.

Consider some canonical examples illustrating different patterns of parallelism. A simple example of pipeline concurrency is the prime sieve. To generate all the prime numbers, one could generate all numbers and remove multiples of 2,3,5,7,..., up to the largest prime computed thus far. As soon as a number is identified as prime, it is added to the sieve and numbers are also eliminated by testing for divisibility by this prime (see Figure 1).

The earlier stages of this particular pipeline are a bottleneck because many more numbers are divisible by smaller primes. The linear pipeline can be improved by changing it to a tree with the numbers sent to different identically behaving objects, each testing for divisibility by a given (low) prime, and then merging the results. This can be achieved by using demand-driven evaluation which dynamically creates context objects to filter the numbers for divisibility of

the primes below it. Specifically, each number can create its own copies of the elements of the sieve as it goes along. This scheme provides tree pipelining for testing divisibility.

It should be observed that because a large number of unnecessary tests are performed, the technique of generating all numbers and then filtering is quite inefficient in the first place. An improved version would avoid generating multiples of the low primes, 2,3,5,.. (up to some prime). Athas describes the behavior of an algorithm designed precisely to do this [7].

Divide-and-conquer concurrency algorithms can often be expressed as functions. Arguments to a function are evaluated concurrently and their values collected to determine the final result. Consider the problem of determining the product of a list of numbers [4]. We can represent the list as a tree as in Figure 2. The problem can be recursively subdivided into the problem of multiplying two sublists, each of which is concurrently evaluated, and their results are multiplied (see Figure 3). The prod-

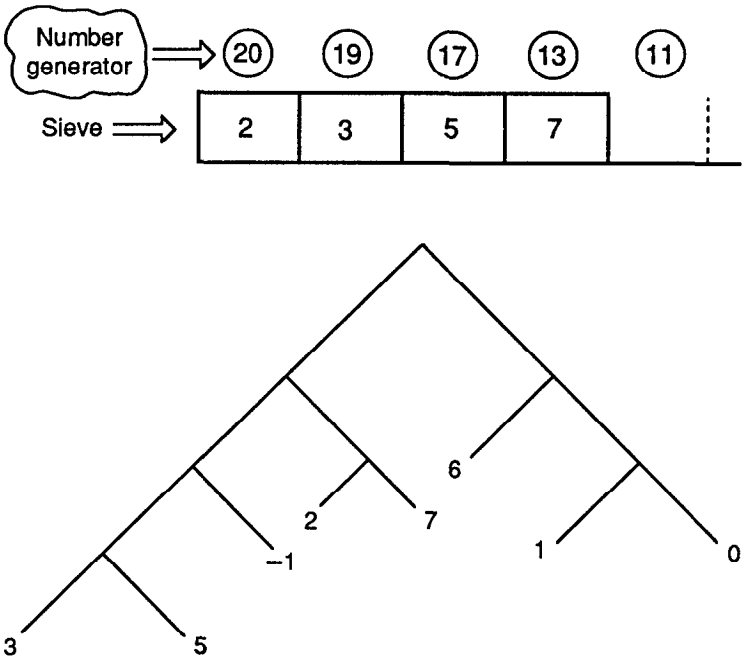

FIGURE 1. ***A simple prime sieve.*** **Numbers generated and successively tested for divisibility by a linear pipeline of primes. The circled numbers represent numbers which are being generated. The boxed numbers are eliminating numbers which are not prime. New primes are added at the end of the sieve.**

FIGURE 2. **A list of numbers to be multiplied. The numbers are represented as leaves of a tree.**

```
(define tree-product
   (lambda [tree]
      (if (number? tree)
         tree
         (* (tree-product (left-tree tree))
         (* (tree-product (right-tree tree))))))
```

FIGURE 3. **The code for multiplying a list of numbers represented as a tree. In the above code, a tree is passed to `tree-product` which tests to see if the `tree` is a number (i.e. a singleton). If so it returns the tree, otherwise it subdivides the problem into two recursive calls. `left-tree` and `right-tree` are functions which pick off the left and right branches of the tree. Note that the arguments to * may be evaluated concurrently.**

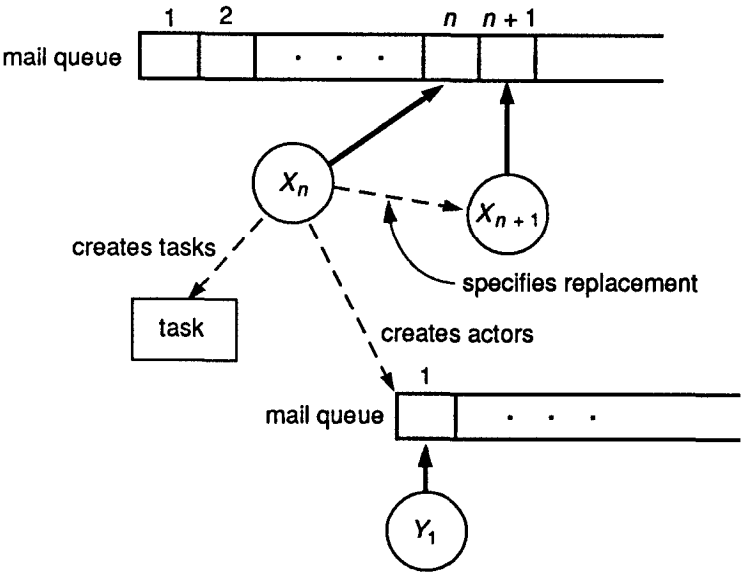

FIGURE 4. **A pictorial representation of the behavior of an actor. When the actor processes the n^{th} communication, it determines the behavior which will be used to process the $n+1^{st}$ communication. The mail address of the actor remains invariant. The actor may also send communications to specific target actors and create new actors.**

uct is then returned. The tree product program in Figure 3 looks very much like code in Lisp or Scheme except that the evaluation strategy is maximally concurrent.

In cooperative problem solving concurrency, intermediate results are stored in objects and shared by passing messages between objects. Simulation programs, where a logical object represents a physical object, is one application of this kind of concurrency. For example, the dynamic evolution of the paths of a number of bodies under the influence of each others' gravitational fields can be modeled as systems of cooperating objects. Another example of cooperative problem solving is blackboard systems which allow collaboration between agents through a shared work space. In an object-based system, the blackboard and the agents may be represented as systems of objects.

The Actor Model

A common semantic approach to modeling objects is to view the behavior of objects as functions of incoming communications. This is the approach taken in the actor model [21]. Actors are self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing. The basic actor primitives are (see Figure 4):

`create`: creating an actor from a behavior description and a set of parameters, possibly including existing actors;

`send to`: sending a message to an actor; and

`become`: an actor replacing its own behavior by a new behavior.

These primitives form a simple but powerful set upon which to build a wide range of higher-level abstractions and concurrent programming paradigms [3]. The actor creation primitive is to concurrent programming what the definition of a lambda abstraction is to sequential programming (For example, see [1]): it extends the dynamic resource creation capability provided by function abstractions to concurrent computation. The become primitive gives actors a history-sensitive behavior necessary for shared mutable data objects. This is in contrast to a purely functional programming model and generalizes the Lisp/Scheme/ML sequential style sharing to concurrent computation. The send to primitive is the asynchronous analog of function application. It is the basic communication primitive causing a message to be put in an actor's mailbox (message queue). It should be noted that each actor has a unique mail address determined at the time of its creation. This address is used to specify the recipient (target) of a message.

In the actor model, state change is specified using replacement behaviors. Each time an actor processes a communication, it also computes its behavior in response to the next communication it may process. The replacement behavior for a purely functional actor is identical to the original behavior. In other cases, the behavior may change. The change in the behavior may represent a simple change of state variables, such as change in the balance of an account, or it may represent changes in the operations (methods) which are carried out in response to messages.

The ability to specify a replacement behavior retains an important advantage over conventional assignment statements: assignments to a variable fix the level of granularity at which one must analyze a system. By contrast, the replacement mechanism allows one to aggregate changes and avoid unnecessary control flow dependencies within computational units which are defined by *receptionists* [2]. Replacement is a serialization mechanism which supports a trivial pipelining of the replacement actions: the aggregation of changes allows an easy determination of when we have finished computing the state of an actor and are ready to take the next action. For example, suppose a bank account actor accepts a withdrawal request. In response, as soon as it has computed the new balance in the account, it is free to process the next request—even if other actions implied by the withdrawal request are still being carried out. To put it another way, the concurrent specification of replacement behaviors guarantees noninterference of state changes with potentially numerous threads running through an actor under a multiple-readers, single-writer constraint.

Concurrent computations can be visualized in terms of event diagrams (see Figure 5). These diagrams were developed to model the behavior of actor systems. Each vertical line, called a *lifeline,* represents all the communications received by a given actor. The receipt of a communication represents one kind of event. Another kind of event is the creation of a new actor represented by an open arc on the top of a lifeline. Connections between lifelines represent causal connections between events. Pending events, representing communications which have been sent but not received, may be represented by activation lines whose arrows note the message and the target.

Control Structures

Concurrent control structures represent particular patterns of message passing. Consider the classic example of a recursive control structure which illustrates the use of customers in implementing continuations. The example is adapted from [14] which provided the original insight exploited here. In a sequential language, a recursive formula is implemented using a stack of activations. There is no mechanism in the sequential structure for distributing the work of computing a factorial or concurrently processing more than one request.

Our implementation of the factorial actor relies on creating a *customer* which waits for the appropriate communication, in this case from the factorial actor itself. The factorial actor is free to concurrently process the next communication. We assume that a communication to a factorial

includes a mail address to which the value of the factorial is to be sent. In response to a communication with a non-zero integer n, the actor with the above behavior will do the following:

- Create an actor whose behavior will be to multiply n with an integer it receives and send the reply to the mail address to which the factorial of n was to be sent.
- Send itself the "request" to evaluate the factorial of $n-1$ and send the value to the customer it created.

One can intuitively see why the factorial actor behaves correctly, and can use induction to prove that it does so. Provided the *customer* is sent the correct value of the factorial of $n-1$, the customer will correctly evaluate the factorial of n. Moreover, the evaluation of one factorial does not have to be completed before the next request is processed; (i.e., the factorial actor can be a shared resource concurrently evaluating several requests). The behavior of the factorial actor in response to a single initial request is shown in Figure 6.

This particular function is not very complicated, with the consequence that the behavior of the customer is also quite simple. In general, the behavior of the customer can be arbitrarily complex. The actor originally receiving the request delegates most of the processing required by the request to a large number of actors, each of whom is dynamically created. Furthermore, the number of such actors created is in direct proportion to the magnitude of the computation required.

There is nothing inherently concurrent in the recursive algorithm to evaluate a factorial. Using the algo-

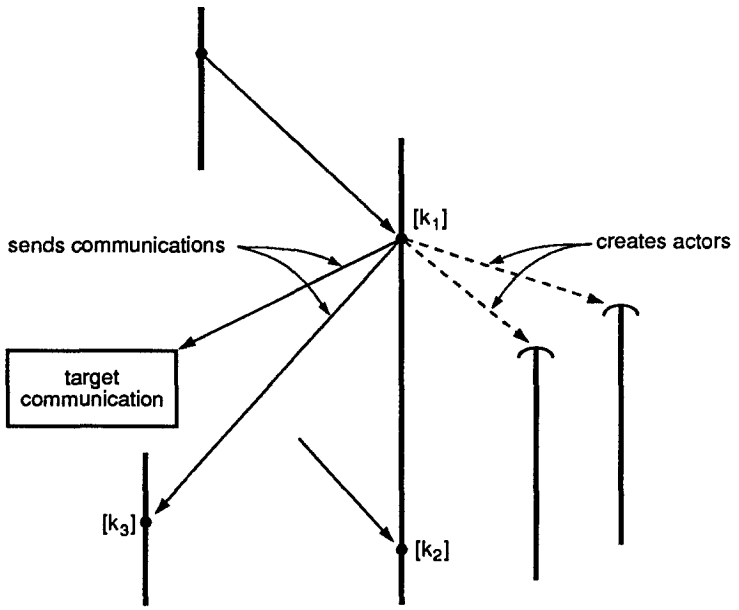

FIGURE 5. Actor event diagrams provide an abstract view of computation in a concurrent system. An actor is identified with a vertical line which represents the linear arrival order of communications sent to that actor. In the above diagram, k_1 and k_2 represent two communications sent to the actor α. The communication k_1 arrives before k_2. In response to processing a communication, new actors may be created (dashed lines) and different actors may be sent communications (solid lines) which will arrive at their target after an arbitrary but finite delay. The box represents a communication which has been sent but not yet processed.

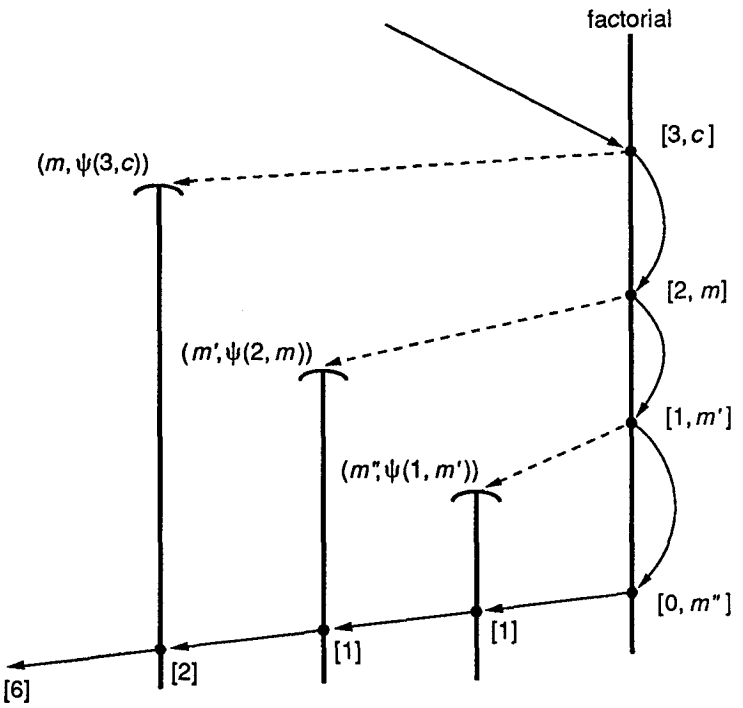

FIGURE 6. *A recursive factorial computation.* The computation is in response to a request to evaluate the factorial of 3. Each actor is denoted by a mail address and a behavior. The ψ's represent the behavior of the dynamically created customers. For example, the behavior ψ(3,c) sends [3*k] to the mail address c in response to the communication k.

rithm in Figure 6, computation of a single factorial would not be any faster if it were done using an actor language as opposed to a sequential language. All we have done is represent the stack for recursion as a chain of customers. However, given a network of processors, an actor-based language could process a large number of requests much faster by simply distributing the actors it creates among these processors. The factorial actor itself would not be as much of a bottleneck for such computations.

Patterns of communications represented in recursion, iteration, divide and conquer, etc., can be abstracted into linguistic forms which automatically coordinate independent computations. An important service provided by high-level actor languages such as Acore [20] is the generation and coordination of *customers* which are actors provided in a request message. A customer can be sent a *reply* message when a *request* is completed, or a *complaint* message if it is not possible to successfully complete the request.

History-Sensitive Behavior

Often it is necessary for an actor to change its local state and to respond to more than one kind of message. For example, a bank account changes its behavior in response to processing an incoming deposit or withdrawal message. In order to define these kinds of actors, a form called mutable is provided in the Rosette actor language developed at MCC by Tomlinson and others in collaboration with the author. The mutable form is used to define a generator actor which creates actors using a behavioral template.

The behavior of a simple bank account may be defined using a mutable expression (see Figure 7). The generator actor that results from the mutable expression is bound to the symbol BankAccount. The generator allows creation of instances via a create expression. Following the keyword mutable is a sequence of identifiers for the state variables of an instance of BankAccount. In this case there is just a single state variable, balance. The methods or communication handlers associated with a BankAccount follow. A method is specified by listing a keyword representing the operation to be executed by the method, followed by a table that represents the content of the request message (in this case a withdraw-from message must specify an amount), and a body that defines how such messages are to be processed. When used within the body of a mutable generator, the form become is used to specify the replacement behavior of the instance of an actor created using the generator—the generator itself does not change.

A new BankAccount may be generated with an initial balance of 1000 by using the create operation as follows:

```
(define my-account
 (create
  BankAccount 1000))
```

We can make the communication handles (the keywords determining which method is to be performed) visible by declaring them to be operations. The behavior of an operation is to send to its first argument a message containing itself and the rest of the arguments it received. This allows object-oriented message-passing style and functional styles to be freely mixed in an actor language. The two styles serve as duals of each other.

Requests may be issued to the new account by using the methods that are declared as operations:

```
(deposit-to my-account 100)
                ⇒ 'deposited 100

(withdraw-from my-account 78)
                ⇒ 'withdrew 78
```

Subexpressions are evaluated concurrently. Thus, the computation of a replacement behavior, done by the become command, is concurrent with the computation of a response. The capability to access the account may be passed to another actor, dynamically reconfiguring a system; for example:

```
(send-to my-wife my-account)
```

would allow the actor whose mail address is bound to `my-wife` to access the actor `my-account`.

Join Continuations

Divide and conquer concurrency can often be naturally expressed by using a functional form which evaluates its arguments concurrently. Implementation of such forms requires the specification of a join continuation which synchronizes the evaluation of the different arguments. For example, the `tree-product` program given in Figure 3 can be expressed in terms of actor primitives as shown in Figure 8.

The form `(bar foo)` represents an asynchronous message `foo` sent to `bar`. The form `[x1 x2...]` represents a data constructor whose elements `x1, x2,...` are concurrently evaluated. It should be noted that the two tree products in the do body are also concurrently evaluated and their values are sent to `new-cust`, where `new-cust` is a history-sensitive actor whose role is to store the first value it receives and multiply the stored number with the second number it receives in order to produce the final response that is sent to the customer which was specified at the time of its first invocation. The form `rlambda` defines actors rather than actor behaviors (which are defined by mutable). In the body of an rlambda, become specifies the replacement behavior of the actor itself. Thus rlambda is the actor analogue of lambda; it captures the history-sensitive behavior of an actor using the form become. Note that in join-cont, v1 refers to the first message received by the actor—which could correspond to either the product of the left subtree or the right subtree.

The behavior of tree-product is shown in terms of an event diagram in Figure 9. When the tree-product actor receives a list represented as a tree containing ltree as its left subtree and rtree as its right subtree, it creates a customer, called a join continuation, which awaits the computa-

tion of the products of each of the two subtrees. The join continuation then proceeds to multiply the two numbers and send the result to the original requester. Because multiplication is commutative, we need not be concerned about matching the responses to the order of the parameters. If we were dealing with an operator which was not commutative, we would need to tag the message corresponding to each argument and this tag would be returned with the response from the corresponding subcomputation. The replacement behavior of the join continuation would then depend on the order in which the results of the evaluation of arguments were received. Because the semantics of concurrency requires that the evaluation of the two invocations of tree-product be indeterminate, the behavior of join-cont cannot be expressed functionally—despite the fact that the behavior of tree-product itself is functional.

In Figure 9, we provide the behavior of the history-sensitive join continuation explicitly. The advantage of explicit join continuations is that they provide considerable flexibility—they can be used to control the evaluation order, to do partial computations, and to do dynamic error handling. For example, if the number 0 is encountered, the join continuation can immediately return a 0—without waiting for the results of evaluating the other subtree [4]. Furthermore, we may want to flag error conditions such as data exceptions. If we require, for example, that all the numbers in the tree be positive

```
(define BankAccount
        (mutable [balance]
                [withdraw-from [amount]
                        (become BankAccount (- balance amount))
                        (return 'withdrew amount)]
                [deposit-to [amount]
                        (become BankAccount (+ balance amount))
                        (return 'deposited amount)]
                [balance-query
                        (return 'balance-is balance)]))
```

FIGURE 7. ***A bank account.*** **The behavior of a bank account is described in object-oriented terms using communication handlers (methods). The behavior is history sensitive: each time a balance query is made, a different response may be given by the bank account.**

```
(define tree-product
    (rlambda [tree customer]
        (if (number? tree)
            (customer tree)
            (let [new-cust (join-cont customer)]
                (do (tree-product (left-tree tree) new-cust)
                (do (tree-product (right-tree tree) new-cust))))))

(define join-cont [custromer]
    (rlambda [v1]
        (become (rlambda [v2]
                (customer (* v1 v2))))))
```

FIGURE 8. **The** `tree-product` **in terms of primitive actors. The behavior of the dynamically** `creation` **join continuation is explicitly shown. We use the form** `rlambda` **to indicate the definition of an actor which may have a replacement behavior. The form** `do` **is used to simply evaluate the two arguments concurrently.**

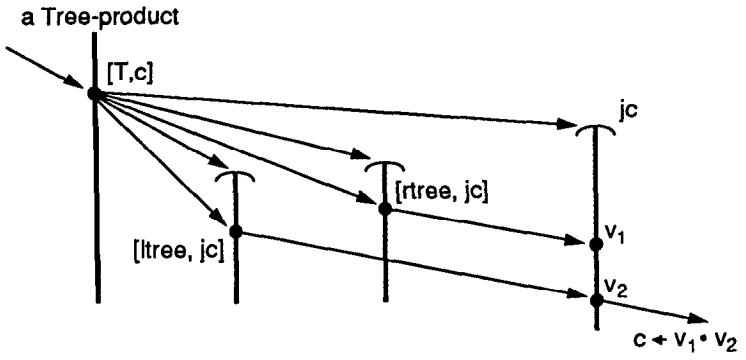

FIGURE 9. ***Tree product event diagram.*** **The skeleton of actions taken by the tree product in response to a message containing the tree T=** `[ltree rtree]` **and the customer** `c`.

```
(define join-cont [customer]
    (rlambda [v1]
        (case
            ((zero? v1) (customer 0))
            ((error? v1) (error-handler v1))
            (otherwise (become (rlambda [v2]
                                (customer (• v1 v2))))))))
```

FIGURE 10. **A join continuation which reacts appropriately to partial computations as they are completed. Note that** `error?` **is an actor which returns a** `true` **or** `false` **value depending on whether the argument it receives is "acceptable."**

we may want to terminate the computation once we encounter a negative number. In this case, we can invoke an error handler to clean up the data or take other appropriate action. Figure 10 specifies the code for a join continuation with such error-handling capability.

In general, error handlers distinct from the regular continuation structures can be passed along. These error handlers provide non-nonfunctional jumps which can appropriately clean-up erroneous conditions that may arise in the course of a computation. Because this separates code with distinct purposes, programming with non-functional jumps supports greater modularity; in particular, it gives us the ability to independently specify and reason about the normal and abnormal behavior of a program.

Nondeterminism

Although actors take a functional view of an object's internal behavior at any given point in time, actors can represent shared history-sensitive objects. Consider the canonical example of a bank account. The behavior of a bank account changes over time as a function of the balance in the account. By contrast, the purely functional programming approach, while mathematically elegant, is insufficient to represent structures in the real world of distributed computing: what makes a shared account meaningful is that state change is visible to many users. The values of functions are, however, returned only to the caller or invoker of that function.

The need to implement synchronization between concurrent computations requires the ability to define an indeterminate, complete merge of messages sent to an actor. We call such a merge a *fair merge.* A fair merge is complete because it merges messages from every sender and may not ignore any sender indefinitely and it is indeterminate because no particular order is specified for messages sent to the same object by different objects. Because shared history-sensitive objects interleave messages sent by different objects, modeling such objects is equivalent to representing a fair merge.

Fair merges are a fundamental concept in modeling concurrent systems; they allow one to abstract over different possible assumptions about the relative speeds of processors, the scheduling of processes on processors, and the relative speed of the communication links. A model which made specific assumptions about such implementation-dependent factors may not be sufficiently abstract to be useful in reasoning about different possible implementations of a concurrent program (see, however, the discussion about coordinated action in section entitled "Coordination"). Using a semantics of fair merge, one can reason about the *eventual* behavior of a concurrent program; reasoning about eventual properties of a concurrent system is analogous to reasoning about fixed points in a recursion in sequential programming.

The behavior of fair merge cannot be represented by the standard *substitution* semantics grounded in the lambda calculus. A substitution semantics requires that an expression have the same value regardless of the context in which it is invoked—a condition violated by the shared bank account whose behavior changes as a function of the balance in the account. Because the ability to implement shared resources is so fundamental to a concurrent system, the actor approach is to integrate the ability to create modifiable, shared structures into the programming model. Furthermore, the connections between objects may be dynamically made or broken. Fair merges in the actor model are implicit—they are captured by the guarantee of message delivery which states that any message sent to an actor must eventually be received—i.e., after a finite but arbitrarily long delay. The guarantee of delivery does not specify the order in which messages may be received by an actor. In particular, the sequence of messages from one actor to another need not be preserved: this allows for the possibility of adaptive routing.

It should be observed that the guarantee of message delivery in an asynchronous communication model itself cannot be realized with certainty—it can only be provided with some level of confidence in a well-engineered system. For example, in principle, an actor could produce enough communications to exceed the buffering capacity of the communication network. This problem is similar to that of implementing recursion using a stack: a recursion may be prematurely terminated by limitations of stack size. In a sequential system, only bounded stacks are physically realizable but the bounds would vary with every specific implementation.

Building Actor Systems

Computation in a typical actor system is performed by the decisions and communications of many small modules. Actor primitives provide a very low-level description of concurrent systems—much like an assembly language. Higher-level constructs are necessary both for raising the granularity of description and for encapsulating faults. Such constructs delimit computational boundaries by aggregating a collection of events into organizational units characterized by patterns of interactions. Simple examples of patterns of interactions include control structures, synchronous communication, and transactions. The following discussion elaborates on these examples and their use in implementing actor systems.

Coordination

A simple example of coordination is that required in function calls which return a value to the (usually implicit) join continuation which is the return address for the call. The creation of join continuations increases the available concurrency in a function call. Thus, function calls are an example of a very simple two-party interaction involving synchronization between a message and an actor.

A more complex example of a two-party interaction is synchronous communication between two actors; such communication represents an

atomic interaction between two actors. Because there is no instantaneous action at a distance in a distributed system, synchronous communication can be implemented only by strongly constraining the implementation. In particular, its implementation requires a known time bound on the potential communication delay between the two actors which can communicate synchronously.

The synchronous communication model is critical to the problem of coordination in distributed systems. Specifically, it allows not only information to be shared but the meta-knowledge that the information has been shared: if an actor X communicates information i synchronously with an actor Y, X knows that Y knows i and Y knows that X knows that Y knows i, etc., ad infinitum. This state of knowledge is called common knowledge; common knowledge is essential for coordination between agents who need to act in concert (see, for example, [21]). In the above example, if X will not act until X knows that Y will also act, then it can be shown inductively that neither actor will act unless they have common knowledge. The notion of common knowledge can be generalized to collections of an arbitrary number of agents. Common knowledge may be achieved through mechanisms representing asynchronous communication which is guaranteed to be delivered within a specified bounded time interval. Note, however, that there is no unique global clock in a distributed system; thus time delays must be expressed in relativistic terms (they may be bounded, for example, using a distance metric). The problem of coordination raises a number of interesting linguistic and semantic issues and is an active area of research in the author's group.

Transactions are another example of n-party interactions. In the context of actor systems, communications can be classified as requests or responses. Each request carries a customer to which a (unique) response is to be sent. A transaction is defined

Multicomputers

Several kinds of concurrent computer architectures have been proposed. These architectures may be broadly divided into synchronous computers, shared memory computers, and multicomputers (also called message-passing concurrent computers). Synchronous computers, such as the Connection Machine, are suitable for data-parallel computation. These computers are quite special-purpose and rather restrictive in their model of concurrency. We are primarily interested in general-purpose computing—for this purpose, computers which support control parallelism are of greater interest.

Shared memory computers have multiple processors and provide a global shared memory. For efficiency reasons, each processor also has a local cache, which in turn creates the problem of maintaining cache coherence. The shared memory computers that have been built typically consist of 16 to 32 processors. Because large numbers of processors create increased contention for access to the global memory, this kind of architecture is not scalable [10].

Multicomputers use a large number of small programmable computers (processors with their own memory) which are connected by a message-passing network. Multicomputers have evolved out of work done by Charles Seitz and his group at Caltech [9] and have been used to support actor languages [7]. The network in multicomputers supports the actor mail abstraction; memory is distributed and information is localized on each computer. Load balancing and maintaining locality of communication simplified by using small objects which can be created and destroyed dynamically, qualities which are characteristic of actor systems.

Configurations of multicomputers with only 64 computers exhibit performance comparable to conventional supercomputers. It should be noted that machines based on the transputer are also multicomputers, but these computers use a model of computation based on communicating sequential processes rather than the actor model.

Multicomputers may be divided into two classes: medium-grained multicomputers and fine-grained multicomputers. Two generations of medium-grained muiticomputers have been built. A typical first-generation machine (also called the cube or the hypercube because of its communication network topology) consisted of 64 nodes and delivered 64 MIPS. Its communication latency was in the order of milliseconds. The typical second-generation medium-grained multicomputer has 256 nodes, can carry out about 2.5K MIPS and has a message latency in the order of tens of microseconds. The development of these machines continues. Third-generation machines are expected to be built over the next five years and increase the overall computational power by two orders of magnitude and reduce message latency to fractions of a microsecond [9].

However, the frontiers of multicomputer research are occupied by work on fine-grained multicomputers. These computers realize an idealized actor machine with fine-grain concurrent structure inherent in the functional form of an actor's behavior; in turn the Actor model is well-suited to programming them (for example, see [10]). Two projects building experimental fine-grained multicomputers are the J-Machine project by William Dally's group at M.I.T. [11] and the Mosaic project by Charles Seitz's group at Caltech. The experimental prototype of the Mosaic system will consist of 16,384 nodes and is expected to deliver 200,000 MIPS [9].

Obviously, Actor languages can be implemented on a number of computer architectures such as sequential processors, shared memory machines, and SIMD architectures. However, multicomputers are particularly interesting because of their scalability characteristics. It should also be observed that actors can be directly supported on multicomputers whereas implementing other programming paradigms on such computers may require their implementation in terms of some simple variant of the actor execution model (For example, see [12]).

as all events intervening in the combined causal and arrival ordering between a given request and a response to it. An event (processing of a message) precedes another event in the causal order if the second event is the result of processing the message corresponding to the first event. The arrival order represents the order in which messages are processed by a given actor.

Transactions delineate computational boundaries for error recovery or for the allocation of resources. Transactions have been used to support debuggers in concurrent computation (see below) and have been proposed as a mechanism for determining resource allocation policies at a high level (see the section entitled "Resource Management").

Visualizing Actor Programs

There are two important difficulties in monitoring and debugging concurrent programs. First, concurrency implies that program execution is nondeterministic. Thus any given execution trace of a program is not likely to be repeated. Second, in any reasonable-sized concurrent system, the large number of objects and interactions between the objects results in enormous data consisting of the large set of events and their relations. This is further complicated by the fact that since there is no unique or true global state in a distributed system, the observations of on-going activity in a system themselves suffer from nondeterminism due to the observer's frame of reference.

One approach to monitoring computations is to retroactively reconstruct relations between events which have already occurred; this can be accomplished by constructing event diagrams of the sort described in Figure 5. Each actor records the communications it has received in the order it receives them and the actions it takes in response to those communications—namely, the communications it sends out and the actors it creates. The sending and receiving events are linked by looking at the recordings in the actor which was the target (specified recipient) of the communication.

A problem with the use of event diagrams is that they contain every event and, in any realistic concurrent system, there are simply too many events. Using event diagrams to try to pinpoint an error can be harder than looking for a needle in a haystack. Mechanisms are necessary to structure events by delineating computational boundaries. One such mechanism is based on the concept of transactions. For our purposes, a transaction is delimited by events between a request message and a response message, where "between" is in terms of a partial order defined by the transitive closure of causal and arrival orders on events. In addition, the transaction must obey the following properties:

1. if a request generates subrequests, the corresponding subtransactions must also be within the transaction
2. no subtransaction is shared with other transactions, (i.e., two independent transactions include the same subtransaction).
3. the first two conditions recursively hold for the subtransactions.

Transactions can be used to pinpoint errors in much the same way a microscope is used to pinpoint areas by recursively focusing on smaller regions which contain the potential point of interest (see Figure 11). In case of a transaction which does not meet its specification, one can look at its subtransactions to determine which of these subtransactions may be causing the observed error. In standard usage, the term transaction also implies a requirement of atomicity, (i.e., either all events that are part of a transaction occur or none of them occur). While the atomicity requirement is useful for delimiting boundaries for error recovery, it is often too strong for a general-purpose programming language.

A debugging system based on this mechanism has been implemented by Manning at MIT. The system is called the observatory [19]. When the transactional structure is not preserved in a computation, for example because of a shared subcomputation, the request reply structure still serves as a granularity control mechanism. Using the observatory, one can end up potentially weaving back and forth through connected computations looking for the source of error. In order to address this kind of difficulty, Yonezawa's group at the University of Tokyo has developed an alternative method for monitoring programs based on collecting objects into groups (see [24]).

Resource Management

In a concurrent language, a problem is often dynamically partitioned into subproblems. An application developer needs to be able to specify the allocation of resources to each dynamically created subproblem. Because thousands of subcomputations may be created, such allocation decisions need to be specified at a high level of abstraction. Sponsors are actors which connect to the underlying resource management system and are used to drive or throttle a computational path.

For example, in a graph search problem, examining a node may suggest examining a number of other neighboring nodes. This process may be termed node expansion. All nodes which are candidates for expansion may not be equally promising candidates; thus a sponsor is needed to specify how much resource to allocate to a node expansion. Furthermore, an allocation decision may need to be dynamically modified as a computation proceeds. In order to determine how much resource to give to a node expansion in a graph search algorithm, a sponsor may use a given sponsorship algorithm which measures the goodness of a target expansion. The sponsor may also be a function of current resources which are available for the computation.

A decision about whether to perform a subcomputation is thus specified independently of how the computation itself is to be carried out. Furthermore, a sponsorship algorithm does not need to specify details about how to utilize physical

resources, such as memory or processing power, and does not need to make assumptions about their associated costs. It simply provides very high-level control over the relative rates or extent of unfolding of particular computational paths.

In many computations, it is natural for subcomputations to be shared between a number of independent computations. A number of messages may be merged before being processed; for example, a number of requests to a money market account may be merged into a single subcomputation involving the trading of a set of stocks. In this case, explicit policies must be developed to merge resources and, conversely, to assess computation costs.

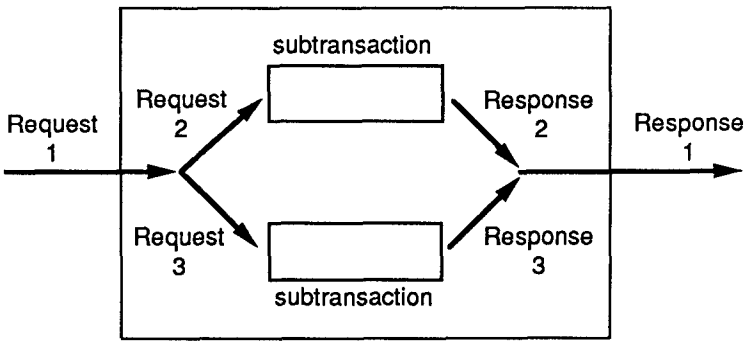

FIGURE 11. A Transactional Structure. Each request generates a unique response.

Other Models of Objects

A number of COOP language models unify a declarative view of objects as abstract data types with a procedural view of objects as sequential processes. In this model, each object is a sequential process which responds to messages sent to that object. Every object may execute its actions concurrently. The number of objects which have a pending message to process (i.e., are potentially active) at a given time during the execution of a program is called its concurrency index at that time. In particular, the concurrency index is limited by the total number of objects in a system at a given time, although new objects may be created dynamically. The concurrency index may be increased by invoking objects asynchronously —thus activating other objects.

Languages which use a process model of objects include ABCL [24], POOL [5], Concurrent Smalltalk (see [25]) and BETA (see [22]). Some other languages realize a simple variant of the Actor model: the body of an object is executed sequentially and assignments may be used within the body but messages are buffered and may not be received during "intermediate" states of an object. Semantically, the behavior of all COOP languages, whether they use a process model or otherwise, can be modeled by actors, much as functions can describe the semantics of procedures in sequential programming.

A traditional sequential process model allows arbitrary control structures to be specified within the body of a given object. The traditional model also encourages sequencing of actions which is potentially unnecessary from the point of view of understanding the concurrency inherent in the logic of a program. An advantage of a process model is that it allows the explicit specification of state change using a familiar one step at a time assignment to variables; the variables are, of course, encapsulated within a given object. Another advantage is that a programmer can optimize the size of the sequential processes to match the optimal size of processes on a given concurrent architecture. Thus, the process size can be determined by a programmer as a function of architectural characteristics such as the costs associated with process creation, context switching and communication.

On the other hand, the sequential process model of objects has at least three disadvantages. First, sequential processes which are optimal on one architecture may not be so on another with different characteristics. Second, because not all state change is due to the logic of an algorithm, it becomes harder to reason about the parallelism in a particular algorithm. Finally, because assignments of values to variables are frequently used in the sequential process model, it complicates programs by discouraging the use of functional components when such components are adequate.

An alternative approach to providing efficient execution on concurrent architectures is to throttle the concurrency in an inherently concurrent language, using translators which are optimized for a particular architecture. This is an area of active research in actors (as well as in the concurrent implementation of declarative languages such as functional programming languages and the so-called concurrent logic languages).

Inherent Concurrency

There are some basic language design decisions involved in providing a notation to specify the behavior of objects; these decisions affect what kind of concurrency can be extracted from the behavioral description of an object. In particular, two styles of expression evaluation can be identified:

Call/Return Style. Subexpressions in the code are evaluated (possibly concurrently) and their values are substituted before proceeding to the enclosing expression.

Customer-Passing Style. Subexpression's evaluations and the join continuations' creation are initiated concurrently. The object is then free to accept the next message. A join continuation takes the results of subexpression evaluations and car-

ries out the rest of the computation specified by the original computational thread provided in the object's behavior.

The customer-passing style supported by actors is the concurrent generalization of continuation-passing style supported in sequential languages such as Scheme.[2] In case of sequential systems, the object must have completed processing a communication before it can process another communication. By contrast, in concurrent systems it is possible to process the next communication as soon as the replacement behavior for an object is known.

Note that the ability to distribute work in systems using call/return style, those using customer-passing style, and those that do or do not sequence actions within objects, may be identical. For example, the language Cantor developed at Caltech, uses sequential execution of code in the body of an object. Cantor has the full power of actor languages; it supports dynamic creation of objects, asynchronous message passing between objects, and atomic replacement behaviors. In case of primitive actor actions, typically asynchronous message sends, sequencing actions within an object causes minimal delay; the time required for these actions is fairly small and the resulting activity is concurrent. However, when arbitrarily complex expressions are to be evaluated, unnecessary sequential dependencies can create significant bottlenecks.

Consider the concurrent implementation of the mergesort algorithm. Assume we have a linked list of numbers which we want to sort. Of course, a linked list is a (very) sequential data structure; we are using it here for illustrative purposes only. A linked list can be split in $n/2$ steps where n is the length of the list, provided that n is known—essentially we have to walk the pointer links to the middle of the list and create a pointer (call it second) to that part of the list. If the length of the list is not known, it would take an extra n step to determine it.

After a list is split, the two sublists can be concurrently sorted using mergesort, and the results merged by successively comparing an element from each of the two lists and picking the smaller one. The next element from the list to which the lesser element belongs is then used in the next comparison. Thus, given two sorted lists, merge produces a sorted list containing elements in both lists. It should be noted that this merge procedure has nothing to do with the concept of merge in concurrency which represents an interleaving of all incoming messages discussed earlier.

The mergesort algorithm can be expressed as in Figure 12.

The form `let*` represents multiple (possibly recursive) let bindings and first-half and second-half return the respective halves of the list and their lengths. As recursive calls are made, the list is split until we have singletons. Each split requires half the number of operations of the previous. As in a sequential mergesort, the total number of operations is $O(n \log n)$; however, the concurrency index doubles each time a split is made. Thus the splits can potentially be executed in $O(n)$ time—given a sufficient number of processors and assuming constant overhead. Initially there are $n/2$ merges involving only two elements and these can be carried out concurrently. The final step involves a single merge of two lists of roughly $n/2$ elements. The merges takes $O(n)$ time since in the final merge one has to walk down the two lists doing comparisons.

Following Athas [17], Figure 13 gives the concurrency index (CI) for the mergesort algorithm executed on 1000 elements. To simplify counting execution steps, we assume all processes are run synchronously—although the algorithm has no synchronous processing requirement. Each interval in the x-axis of the diagram represents the processing of a single message by all actors which have a pending message. Furthermore, message delivery is assumed to take one time step. These time steps are called sweeps. Notice that a difficulty with this algorithm is that it requires roughly n processors to sort a list of n numbers. However, most of these processors would be idle much of the time (as far the execution of the mergesort algorithm is concerned). In practice, the processing corresponding to a sweep will be delayed until all the processing in the previous sweep can be completed. In other words, the concurrency index curve plotted as a function of steps needs to be truncated at the maximum number of processors available. Thus, executing the algorithm on a real parallel computer will take at least a time factor which equalizes the areas under the two concurrency index curves (the truncated curve and the curve assuming a sufficiently large number of processors).

The total time efficiency of mergesort in the presence of a limited number of processors can be improved by the following observation: Because the beginning element and length of the first half of the list are known, the first half of the list is determined even as the first element of the second half is being computed.[3] Thus, one can start sorting the first half of the list concurrently with computing the first element of the second half of the list. The algorithm in Figure 14 provides a skeleton of how this can be done.

Figure 15 plots the expected ideal behavior of this algorithm (as simulated by the Rosette system). It gives the concurrency index as a function of the number of sweeps. Note that the processors are more uniformly busy and the maximum number used is only a small fraction of the number of elements in the list. The reason for the more uniform concurrency index is that the concurrency index builds up much more rapidly as more mergesorts are triggered.

The ease of expression in an inher-

[2]It is interesting to note that Scheme itself was inspired by an attempt to understand the concept of actors as it was first proposed [1].

[3]This observation was communicated to the author by Chris Tomlinson.

ently concurrent language simplifies noting the data dependencies which are simply expressed as synchronizations implicit in function calls. On the other hand, it is possible to express the same code in terms of sequentially executed primitive actor bodies—without any meaningful loss of speed. The second case requires that, instead of waiting for an arbitrarily large number of objects to execute, the dynamic creation of a number of context objects be explicitly specified to carry out the subcomputations. In an inherently high-level actor language, this work is simply transferred to a compiler.

Communication and Coordination

Because there is no instantaneous action at a distance, the interactions between components of a distributed system must be built in terms of asynchronous communication. In an asynchronous communication model, a sender is free to take further action after dispatching a given message. This is in contrast to sequential object-oriented languages such as Smalltalk which use synchronous communication: in this case, a sender waits for a response before continuing its execution. Synchronous communication fits naturally with the use of a single active computational thread which weaves through different objects which are invoked and return a value to their caller.

In the context of concurrent computing, however, synchronous communication reduces the number of objects that may be potentially active at any given point in time. If an asynchronous communication model is used, as in the primitive Actor model (see section entitled "The Actor Model"), then *customers* representing return addresses must be explicitly supplied. For example, the behavior of an actor x in response to a [+ 3 c] message may be to return 8 to the customer c which has been supplied in the message. It is often convenient to assume that the value of a subexpression will be automatically substituted for the arithmetic subexpression when the expression is evaluated. Such a notation abstracts from the explicit synchronization which must be implemented in terms of asynchronous message-passing. The difference between implicit and explicit synchronization is similar to the difference between an assembly language and a (sequential) programming language with expressions. The constructs of the actor model are primitives which can

```
(define mergesort
    (rlambda [list len]
        (if (= len 1)
            list
            (let* [first(first-half list len)]
                  [second (second-half list len)]]
                (merge (mergesort first) (mergesort second))))))
```

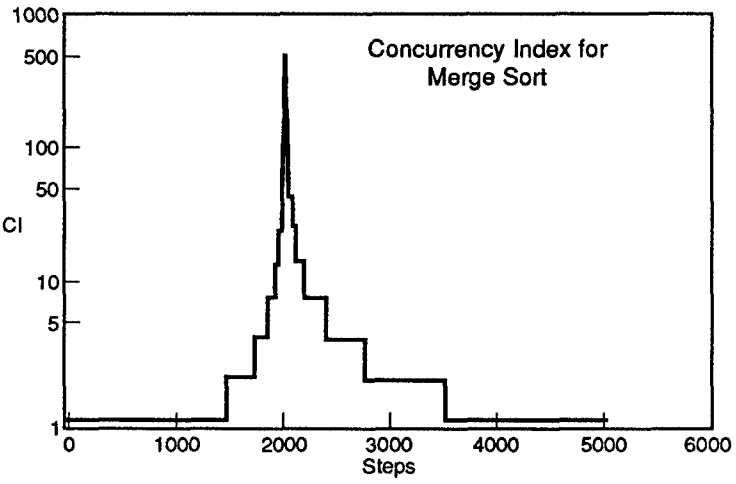

FIGURE 12. ***A concurrent mergesort example.*** **The actor checks to see if the length of the list is 1, if so returns the single number in the list. Otherwise it subdivides the list into two halves which are bound to `first` and `second`. These two halves are sorted using two concurrent recursive calls to `mergesort` and the results sorted lists are merged using the actor `merge`. The example is discussed in [7]. Note that the let bindings are evaluated before the let body.**

FIGURE 13. ***Concurrency Index for a mergesort of 1000 elements.*** **Notice that the vertical axis of the graph is on log scale (from [7]). The horizontal axis represents time steps (sweeps) in which every actor processes a single message. This synchronous processing assumption is used only to simplify counting steps and is not required by the algorithm. The first 1000 steps involve determining the length of a given list by walking through it.**

```
(define mergesort
    (rlambda [list len]
        (if (= len 1)
            list
            (merge (mergesort (first-half [list len]))
                   (mergesort (second-half [list len]))))))
```

FIGURE 14. Code for a concurrent mergesort with with fewer data dependencies. Note that the sequential let has been removed.

be used to build higher-order concurrent procedural and data abstractions.

Reasoning about Object Behavior

Because the state of the components of a system is constantly changing, it is generally impossible to predetermine precisely what state a particular component will be in when another component attempts to interact with it. In models of concurrency based on communicating sequential processes, the effect of a message received at all potential entry points within a process must be considered. In a shared variable model, even more interactions are possible as different information may be written into each shared variable of a process by any other process, creating an exponential number of possibilities for interaction. Each interaction corresponds to a different indeterminate execution of the system.

Actors encapsulate operations so that they may be externally invoked at only at one entry point. This can be achieved by breaking up a sequential process into a number of smaller independent objects; in fact, it is sometimes possible to use formal transformation rules to automatically decompose processes into objects with a single entry point as proposed by Shibayama at the Tokyo Institute of Technology (see [24]). Intermediate states of an actor are invisible to the outside, and actors may not be interrupted in such states. Because interactions between intermediate states of two actors need not be considered, such decomposition promotes modularity; specifically, it can reduce the complexity of invariant properties to be established in order to reason about the behavior of a program. The distinction between actors and procedures with multiple communication entry points can be appreciated by considering their analogy to the difference between procedure calls and unrestricted goto's.

The behavior of an actor is atomic, (i.e., internal loops are prohibited within actors). Thus the interaction problem is transferred to another level: the number of interactions between independently triggered computations can again be large as an actor interleaves messages triggered by distinct requests from different senders. This potential disadvantage is mitigated by two factors: first, actor languages encourage greater use of functional components which are referentially transparent and easy to reason about. Second, actor languages use a customer-passing style to separate the continuation representing an actor's future behavior and the continuation of a computational thread. In other models of concurrent objects, multiple entry points are possible when synchronous communication is used. However, even in some of these models, the target object is invoked at a single entry point and provides a response to the caller at the point of the call.

A second advantage of COOP is the locality properties in the model. An object may send a message only to those objects it knows about [15]. The axioms governing which objects are known to an object are called locality laws; these laws were developed in the context of the Actor model by Hewitt and Baker at MIT. Locality laws further restrict the number of possible interactions between objects which have to be considered in a given system. Locality laws make it possible to model open systems, (i.e., evolving systems which are open to interaction with their environment). Because the outside environment is dynamically changing and may contain unknown elements, its behavior cannot be completely predicted. Therefore an open systems model must allow local reasoning about a module in different possible contexts. By regulating interactions with other actors, locality laws make such local reasoning feasible. In particular, because the constituents of the distributed system will not be known to a single object in the system, (global) broadcasting is not generally a meaningful construct in open systems.

Object-Oriented Programming

Object-oriented programming supports the reusability of code, thus supporting an evolutionary programming methodology, and it provides modularity in programming, thus allowing a separation of design concerns. These powerful aspects of object-oriented programming can be utilized in concurrent programming by providing mechanisms such as inheritance and reflection. This section discusses the basic constructs which can be used to build these mechanisms and points to some interesting research issues.

Inheritance

A powerful feature of object-oriented languages is inheritance. Inheritance was introduced in Simula primarily as a organizational tool for classification. In Simula, objects can be defined as members of a *class* and, as a consequence, share procedures that are applicable to all members of the class; note that members of a class may themselves be classes. Class-based sharing naturally promotes modularity in the code by putting all the code common to a number of objects in one place. Modifying and debugging programs is simplified by making changes to a class whose behavior in turn is visible to all its members. Organization of objects using classification incorporates objects into a tree-like structure and can provide clarity in knowledge representation—as experience in chemistry (periodic table) and biology (taxonomy of species) has shown.

Inheritance essentially makes the code in one object (a class) visible to another object (a member). Code sharing leads to namespace management issues—the same identifier may be bound to procedures and parameters in an object and in its class. Object-oriented languages differ regarding how such name conflicts are handled. In Simula, superclass identifiers are renamed; this essentially provides static bindings which are resolved lexically. Simula also provides for a virtual declaration which allows identifiers

representing variables (but not those bound to methods) in a superclass to be visible in a subclass—a case of dynamic scoping. Virtuals are used to support incomplete specifications which are to be added to by the subclasses. While Simula was not designed to support concurrency, one of the designers of Simula, Kristen Nygaard at the University of Oslo, has developed in collaboration with Ole Madsên at Arhus University and others, the language BETA which explicitly supports concurrent programming. Concurrency in BETA is obtained by explicitly specifying alternation or multisequential execution (see the chapters on BETA in [22]).

By contrast, Smalltalk takes a more operational view of inheritance. Conflicts in identifiers are resolved dynamically to provide greater flexibility. This emphasis led to its use primarily as a programming method to support sharing and reusability of code and data, rather than as a mechanism for classification. A good discussion of a number of issues relating to inheritance and object-oriented programming can be found in [22]. (See, specifically the classification of object-oriented languages developed by Peter Wegner at Brown University.) Related mechanisms include classless schemes such as dynamic inheritance and delegation.

One proposal, advanced by Jagannathan and the author, is to allow programmers to define different possible inheritance mechanisms in a single linguistic framework [17]. Associated with an object is a local environment which provides bindings for the identifiers in that object. The idea is to provide the ability to reify environments (see the discussion in the following section and to explicitly manipulate them as first class objects). For example, two environments may be composed so as to shadow the bindings in one object (for example, a class) with the bindings in another object (for example, an instance). By using different possible compositions, distinct inheritance mechanisms can be obtained and these mechanisms can coexist in the same system. This proposal is an extension of the work of Jagannathan on first-class environments [16].

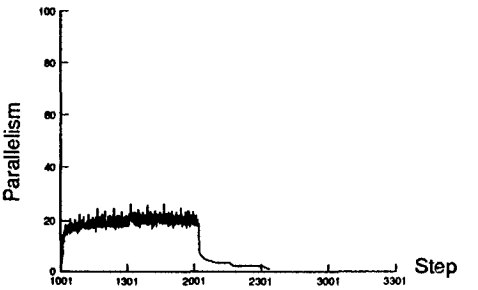

FIGURE 15. Mergesort with fewer dependencies. The concurrency index for a mergesort of 1000 elements is plotted. In this case, only approximately 22 processors are used at any given time. Note that the x-axis begins at 1001 to ignore the sequential walk through the list required to determine its length.

The interaction of concurrency and inheritance raises a number of interesting issues. For example, replacement behaviors in actors are specified atomically. If inheritance is defined in terms of a message-passing protocol between one object and another, the task of determining a replacement may be naturally distributed as part of the replacement behavior is determined locally and part in a different object. This is not an issue in a sequential language like Smalltalk which uses synchronous communication with a single active thread: parts of the state of an object are updated through assignments made by the object and other parts of the state may be assigned by its class. Another complication which arises in concurrent object-oriented languages with inheritance mechanisms is the interaction between inheritance and synchronization constraints. This has been a very active area of research and a number of solutions have been proposed (for example, see [23]); we discuss this interaction briefly in the next section.

Reflection

In the normal course of execution of a program, a number of objects are implicit. In particular, the interpreter or compiler being used to evaluate the code for an object, the text of the code, the environment in which the bindings of identifiers in an object are evaluated, and the communication network are all implicit. As one moves from a higher-level language to its implementation language, a number of objects are given concrete representations and can be explicitly manipulated at the lower implementation level. For example, the join continuation actor in Figure 8 is implicit in Figure 3. When the join continuation is made explicit, it can be usefully modified, as we showed in Figure 10.

The dilemma is that if a very low-level language is used, the advantages of abstraction provided in a high-level notation are lost. Alternately, the flexibility of a low-level language may be lost in a high-level language. Moreover, although it is possible for a low-level program to have a model of its own behavior (for example, as in the case of a Universal Turing Machine), this need not always be the case. A reflective architecture addresses this problem by allowing us to program in a high-level language without losing the possibility of representing and manipulating the objects that are normally implicit [18]. Reification operators can be used to represent at the level of the application, objects which are in the underlying architecture. These objects can then be manipulated like any other objects at the higher application level. Reflective operators may then be used to install the modified objects into the underlying architecture. Reflection thus provides a causal connection between the operations performed on this representation and the corresponding objects in the underlying architecture.

In the example of a tree product, the program can be expressed in a high-level language as a functional product expression. However, when needed, its join continuation can be dynamically reified and a new join continuation actor can be installed to perform the necessary synchronization.

In a COOP system, the evaluator of an object is called its meta-object. Reflective architectures in COOP's have been used to implement a number of interesting applications. For example, Watanabe and Yonezawa (see [24]) have used it to separate the logic of an algorithm from its scheduling for the purposes of a simulation: in order to build a virtual time simulation, messages are time-stamped by the meta-object and sent to the meta-object of the target which uses the time-stamp to schedule the processing of a message or to decide if a rollback is required. Thus the code for an individual object need only contain the logic of the simulation, not the mechanisms used to carry out the simulation; the specification of the mechanisms is separated into the meta-objects.

One application of reflection in actor-based systems is to address the problem of synchronization constraints, (i.e., conditions limiting which communications an actor in a given state is able to process). For example, a bounded buffer which is full cannot service requests to enqueue. In some COOP languages, synchronous communication is used to enforce synchronization constraints; the recipient refuses to accept communications which it is not in a state to process. This solution, while quite simple, can reduce the amount of concurrency available in a system by requiring suspension of the execution of actions by a sender until the recipient is ready to accept the message —even if the sender's future behavior does not depend on whether the message has been delivered.

One approach to increasing concurrency in the synchronous communication model is to dynamically create a new object which attempts to synchronously communicate with the target. The original sender is then free to continue its processing. While theoretically feasible, this solution can be inefficient: it may increase the traffic in the communication network as a sender repeatedly tries to transmit the message to an unavailable recipient.

Another solution, used in actor systems, is to let an object explicitly buffer incoming communications which it is not ready to process (i.e., selective *insensitivity*). For example, an actor may need to process messages in the order in which they are sent by a given sender. However, because of adaptive routing, the order in which messages arrive may be different from the order in which they were sent. In this case, messages which arrive out of sequence can simply be buffered until their predecessors have arrived.

The insensitive actor approach has an important deficiency: it fails to separate the question of what order a given set of tasks can be executed in—i.e., the synchronization constraints from the question of how those tasks are to be executed—i.e., the algorithmic structure of actions to be taken. Such a separation would support local reasoning about feasible actions. In the Rosette language, Tomlinson and Singh proposed a reflective mechanism which reifies a mail queue for an actor and modifies the queue's behavior by making it sensitive to enabledness conditions which capture the synchronization constraints of the actor to which the queue belongs [23].

Research Efforts

The development of architectures and systems based on the COOP model is an active area of research around the world. We briefly describe a few of these efforts. This list is by no means complete but gives a flavor for some of the work under way.

In Europe, several large-scale efforts are under way. Under the auspices of ESPIRIT, de Bakker, America and others have worked on the definition of a parallel object-oriented language called POOL [5]. In the POOL object model, each object has a body, a local process, which starts as soon as the object is created and executes in parallel with the bodies of all other objects. Sending and receiving of messages is indicated explicitly within this body in such a way inside every object everything proceeds sequentially and deterministically. In industrial partnership with Philips, the project has also designed a language-driven architecture called DOOM (Decentralized Object-Oriented Machine). DOOM is a parallel machine consisting of a number of processors (100 in the present prototype), each with its own private memory, and connected via a packet-switching network. Many small and several medium-to-large applications have been written in POOL. Example application areas are databases, document retrieval, VLSI simulation, ray tracing, expert systems, and natural language translation.

Another large ESPRIT project, called ITHACA (Integrated Toolkit for Highly Advanced Applications), is working to produce an object-oriented application development environment including a concurrent object-oriented language and database, a Software Information Base (which will store a large collection of classes), a set of tools for browsing, querying and debugging classes, and tools to support interactive application construction from reusable classes. ITHACA is a 5-year, 100 man-year/year (12 million ECU/year) project led by Nixdorf, with Bull, Geneva, and three other European partners. An academic partner in the project is a group under the direction of Tsichritzis at the University of Geneva.

Japan's Ministry of International Trade and Industry recently announced that it will support a Cooperating Agents Project based on COOP. Initial funding level for this seven year project is estimated to be $35 million. Yonezawa at the Univerity of Tokyo, whose group developed ABCL (an actor-based concurrent language), is an academic leader of this effort. A focus of this

project is to apply the work on actors to coordination technology.

In the United States, where a foundation for COOP was provided by the work of Carl Hewitt and associates at MIT, a number of smaller groups are developing COOP systems. Hewitt's group in particular is focusing on open information systems and artificial intelligence applications. Ken Kahn, Vijay Saraswat and others at Xerox PARC are working on a high-level actor programming language called Janus. The author's group at the University of Illinois at Urbana-Champaign is currently working on programming abstractions and language models for dependable concurrent computing. Finally, the United States has a considerable lead in innovative multicomputer architectures inspired by language models closely tied to concurrent object-oriented programming.

Acknowledgments.
I wish to thank Julian Edwards, Suresh Jagannathan, Carl Manning, Jeff Mantei, Satoshi Matsuoka, Oscar Nierstrasz, Vipin Swarup, Peter Wegner, Rebecca Wirfs-Brock and Akinori Yonezawa for helpful comments on drafts of this article. The ideas expressed in this article have benefitted from interactions with a number of other individuals including Carl Hewitt, Alan Perlis, Mark Scheevel, Vineet Singh, Carolyn Talcott, Chris Tomlinson and Becky Will. The work described in this article has been made possible in part by a Young Investigator Award from the Office of Naval Research (ONR contract number N00014-90-J-1899) and in part by an Incentives for Excellence Award from the Digital Equipment Corporation.

References

1. Abelson, H. and Sussman, G.J. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass, 1985.
2. Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.
3. Agha, G. Supporting multiparadigm programming on actor architectures. In *Proceedings of Parallel Architectures and Languages Europe (PARLE '89), vol. II: Parallel Languages, LNCS 366.* Springer-Verlag, New York, 1989.
4. Agha, G. and Jagannathan, S. *Reflection in concurrent systems: A model of concurrent continuations.* Tech. Rep., Dept. of Computer Science. Univ. of Illinois at Urbana Champaign, 1990. To be published.
5. America, P. Issues in the design of a parallel object-oriented language. *Formal Aspects Computing,* 1, 4 (1989), 366-411.
6. Annot, J.K. and den Haan, P.A.M. POOL and DOOM: The object-oriented approach. In *Parallel Computers: Object-Oriented, Functional, Logic,* P.C. Treleaven, Ed. Wiley, 1990, pp. 47-79.
7. Athas, W. *Fine grain concurrent computations.* Ph.D. dissertation, Computer Science Dept., California Institute of Technology, 1987. Also published as Tech. Rep. 5242:TR:87.
8. Athas, W. and Boden, N. Cantor: An Actor Programming System for Scientific Computing. In *Proceedings of the NSF Workshop on Object-Based Concurrent Programming,* G. Agha, P. Wegner, and A. Yonezawa, Eds. ACM, N.Y., April 1989. pp. 66-68. Special Issue of SIGPLAN Notices.
9. Athas, W. and Seitz, C. Multicomputers: message-passing concurrent computers. *IEEE Comput. 9,* 23 (August 1988).
10. Dally, W. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Press, 1986.
11. Dally, W. *The J-Machine: System Support for Actors.* In *Towards Open Information Systems Science,* C. Hewitt and G. Agha, Eds. M.I.T. Press, Cambridge, Mass., to be published.
12. Dally, W. and Wills, D. Universal mechanisms for concurrency. In *Proceedings of Parallel Architectures and Languages Europe (PARLE '89). Vol. II: Parallel Languages.* Springer-Verlag, New York, 1989. LNCS 366. pp. 19-33.
13. Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. *Solving Problems on Concurrent Processors. Vol I, General Techniques and Regular Problems.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.
14. Hewitt, C. Viewing control structures as patterns of passing messages. *J. Artif. Intell. 8,* 3 (June 1977), 323-364.
15. Hewitt, C. and Baker, H. Laws for communicating parallel processes. In *1977 IFIP Congress Proceedings,* IFIP (August 1977) pp. 987-992.
16. Jagannathan, S. A Programming Language Supporting First-Class, Parallel Environments. Tech. Rep. LCS-TR 434, Massachusetts Institute of Technology, December 1988.
17. Jagannathan, S. and Agha, G. Inheritance through Reflection. Tech. Rep., Dept. of Computer Science, Univ. of Illinois at Urbana Champaign, 1990. To be published.
18. Maes, P. Computational Reflection. Ph.D. dissertation, Vrije University, Brussels, Belgium, 1987. Tech. Rep. 87-2.
19. Manning, C. Traveler: the actor observatory. In *Proceedings of European Conference on Object-Oriented Programming.* Springer Verlag, New York, January 1987. Also appears in Lecture Notes in Computer Science, vol. 276.
20. Manning, C. Introduction to programming actors in acore. In *Towards Open Information Systems Science.* C. Hewitt and G. Agha, Ed., MIT Press, Cambridge, Mass, to be published.
21. Moses, Y. *Knowledge in a distributed environment.* Tech. Rep. STAN-CS-86-1120, Computer Science Dept., Stanford University, 1986.
22. Shriver, B. and Wegner, P., Eds. *Research Directions in Object Oriented Programming.* MIT Press, Cambridge, Mass., 1987.
23. Tomlinson, C. and Singh, V. Inheritance and Synchronization with Enabled Sets. In *Proceedings of OOPSLA-89,* (1989). To be published. ACM, New York.
24. Yonezawa, A., Ed. *ABCL: An Object-Oriented Concurrent System.* MIT Press, Cambridge, Mass., 1990.
25. Yonezawa, A. and Tokoro, M., Eds. *Object-Oriented Concurrent Programming.* MIT Press, Cambridge, Mass., 1987.

CR Categories and Subject Descriptors: C.1.2. [**Processor Architectures**] Multiple Data Stream Architectures; D.1.3. [**Programming Techniques**] Concurrent Programming; D.3.1. [**Programming Languages**] Formal Definitions and Theory—semantics, syntax; D.3.3. [**Programming Languages**] Language Constructs

General Terms: Design, Methodology

Additional Key Words and Phrases: Actor model, concurrency, concurrent programming structures, multiprocessor architectures, object-oriented programming, programming language theory

About the Author

GUL AGHA is assistant professor and director of the Open Systems Laboratory in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His current research interests include concurrent programming languages, open system architecture and coordination technology. Author's Present Address: Dept. of Computer Science, 1304 W. Springfield Ave., University of Illinois at Urbana-Champaign, Urbana, IL 61801. agha@cs.uiuc.edu.

© 1990 ACM 0001-0782/90/0900-0125 $1.50