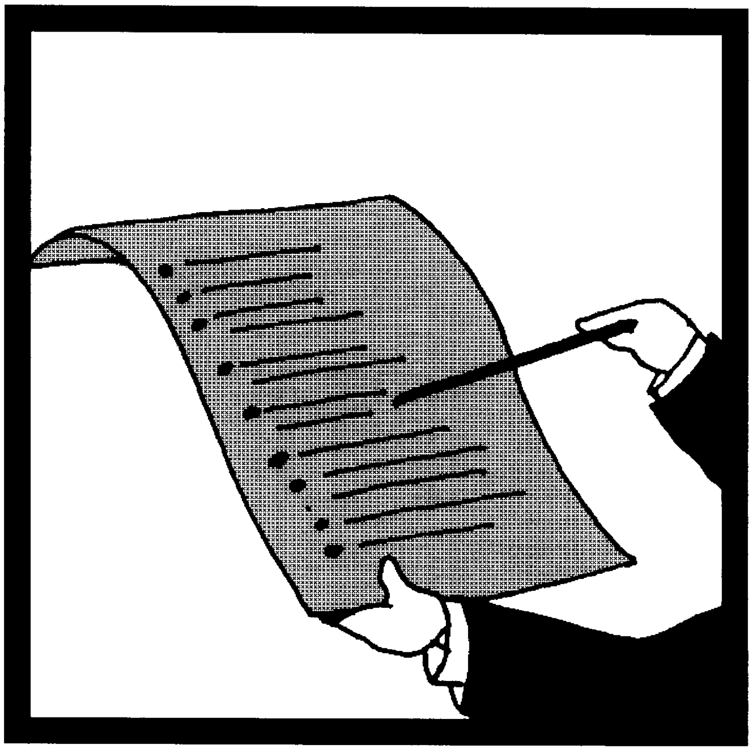
FEATURE

PARALLELISM IN OBJECT-ORIENTED LANGUAGES: A SURVEY

A look at 14 representative languages reveals that when concurrency features are added after a language has been designed, the resulting hybrid can be difficult to use and may produce inefficient programs.

BARBARA B. WYATT
KRISHNA KAVI
STEVE HUFNAGEL
University of Texas at Arlington



Concurrent object-oriented languages try to bring the benefits of object orientation (modularizing a problem into smaller problems based on data rather than function) to multiprocessor environments. We compared how several of these languages deal with communication, synchronization, process management, inheritance, and implementation trade-offs. We also explored how they divide responsibility between the programmer, the compiler, and the operating system. We did not investigate issues unique to distributed process migration, naming, load balancing, or security.

We found that current object-oriented languages that have concurrency features were often compromised in important areas, including inheritance capability, efficiency, ease of use, and degree of parallel activity. Frequently, this was because the concurrency features were added after the language was designed. Unless concurrency, synchronization, and communication are carefully integrated, a parallel object-oriented language can be inefficient and difficult to use.

OBJECT-ORIENTED LANGUAGES

Object-oriented languages break a program down into segments (objects) accessible only by sending messages through a rigid interface. The objects interpret each message and take an appropriate action. Theoretically, you can't access an object's internal data, but the object, in effect, shares this data through its actions.

In these languages, objects inherit features from other objects in a hierarchy. Inheritance classifies objects that share sets of properties. Objects may inherit features from more than one classification structure and more than once from the same class (multiple inheritance). In some cases, objects inherit only a reference to the inherited code instead of a copy of the entire segment.

Inheritance lets programmers reuse code and redefine its application within the current environment. It is the key to building maintainable, reusable systems, and it provides a form of configuration management. Inheritance is one of seven properties that, according to Bertrand Meyer, characterize pure object-oriented languages.[1] The seven properties are

- ♦ modular structure;
- ♦ data abstraction (objects are described as implementations of abstract data types);
- ♦ automatic memory management (the language deallocates unused objects without programmer intervention);
- ♦ classes (every nonsimple type is a class);
- ♦ inheritance (a class may be defined as an extension or restriction of another);
- ♦ polymorphism and dynamic binding (program entities can refer to objects of more than one class, and operations can have different realizations in different classes); and
- ♦ multiple and repeated inheritance (a class can inherit from more than one class and be a parent to the same class more than once).

Meyer regards languages that meet the first four criteria as object-based; he regards as truly object-oriented only languages that meet all seven characteristics.

The hardest requirement to meet is the last one, multiple inheritance. When a child class inherits from two other classes that share a common ancestor, references to that ancestor's methods are ambiguous unless you delineate a specific path. Multiple inheritance makes a language more flexible and expandable, but it is debatable whether or not a language must have this quality to be considered object-oriented. Even the classic object-oriented language, Smalltalk, does not support multiple inheritance.

PARALLELISM

Concurrent languages use constructs for creating processes (like *fork*) and destroying them (like *join*). The operating system optimizes the mechanisms for communication, synchronization, and mutual exclusion according to whether the physical memory is shared or distributed. Memory may be shared by all processors or distributed throughout the system so that each processor has access to only a portion of the memory. Regardless of the physical organization, the logical organization of memory may be either shared or distributed.

Inheritance lets programmers reuse code and redefine its application within the current environment.

In physically shared memory systems, processes communicate by sharing variables, forcing the memory to ensure mutual exclusion by different processes. These systems often use semaphores and spin locks for data synchronization and mutual exclusion.

In physically distributed memories, processes communicate by message passing or remote procedure calls. Blocking and nonblocking message calls can be used for synchronization, but the programmer is responsible for maintaining data consistency when using nonblocking mechanisms.

Concurrent languages in an object environment can allow objects to be created either dynamically as a program runs or only when the program starts. Programmers can synchronize active objects with asynchronous method calls, future variables, and early or late creation of successors. Asynchronous message calls let objects process messages simultaneously without blocking the sending object until no reply is returned to the sender.

Future variables allow similar concurrency, but require a reply. In the future-variable approach, as long as the sender doesn't need the results, both sender and receiver may execute concurrently. Some languages let processing continue before a message has been answered; thus both the sending and the receiving object may be active simultaneously.

Another approach is to allow early creation of the successor (as in Actors-based languages). In the Actors approach, the successor may begin responding to the next message while the parent is still processing its message.[2] Within a single object, it is also possible to create multiple threads of activity either by allowing multiple method invocations in response to a single message or by allowing multiple messages to process concurrently.

In this article, we assume that a programmer is interested in specifying the parallelism; some believe parallelism should be transparent to the programmer.

PROCESS MANAGEMENT

Just as a parallel-processing environment implies multiple processes, a parallel object-oriented system spawns multiple objects, each of which can start a thread of execution. Objects and processes, however, are independent of each other. Processes invoke methods contained in objects. Table 1 shows how some languages compare in their support of process management.

Process creation. Most concurrent object-oriented languages use one of two approaches to start multiple processes. In the explicit approach, the language provides a mechanism for spawning multiple processes, external to the object structure. In this approach, parallelism sits on top of the object structure rather than being integrated into it. Explicit mechanisms like locks, monitors, and semaphores ensure object integrity. This approach can be im-

TABLE 1
PROCESS-MANAGEMENT FUNCTIONS

Language	Creation	Termination	Activation	Granularity
Abcl/1	Implicit	Continue after reply	On message receipt	Medium
Abcl/R	Implicit	Continue after reply	On message receipt	Medium
Actor	Implicit	On reply	On message receipt	Fine
Argus	Explicit	Terminate on reply	Entry calls to guardian	Coarse
Concurrent Smalltalk	Implicit	Continue after reply	On message receipt	Medium
COOL	Implicit	Terminate on reply	Invoking a parallel function	Medium to large
Eiffel	Explicit	Terminate on reply	Invoking process object	Medium
Emerald	Implicit	Continue after reply	On creation	Coarse
Gnu C++	Implicit	Continue after reply	On creation	Coarse
Hybrid	Explicit	Continue after reply	On creation of object and thread	Coarse
Nexus	Explicit	Terminate on reply	On message receipt	Coarse
Parmacs	Static	Continue after reply	On program start	Medium
POOL-T	Implicit	Continue after reply	On creation	Medium
Presto	Explicit	Terminate on reply	On creation of thread object	Large

plemented with a set of predefined threads or root object types for initiating parallel activity.

For example, in Presto,[3] a thread object (which contains a program counter and a stack of method invocations) is the basic unit of execution. Two functions, create and start, control thread execution. This approach makes it easier to add parallel capabilities to an existing language — available compilers and support software need not be modified. The thread and root objects can be included in the program, and inheritance can create different types of thread and root objects.

In the implicit approach, an object invocation can spawn multiple execution threads. In this approach, processes are encapsulated within objects, creating composite objects. When an object gets a message, it can activate one or more internal objects.

Languages that adopt this approach can increase parallelism by creating objects dynamically. The runtime system schedules and controls parallel activity by keeping a list of objects on each available processor that can be run. Rather than being limited by the user's view of the amount of inherent parallelism, the runtime system could face the opposite problem by creating more objects than available resources (although it can combine small objects to increase efficiency). An advantage of this approach is that resources can be allocated more easily in response to changing conditions.

More languages have adopted the implicit approach because it abstracts the details of setting up multiple processes, easing the programmer's task. The explicit approach requires two abstraction levels, objects and threads, blurring the unit of concurrency, and makes it the programmer's responsibility to specify the parallel activity. This is especially hard to do in a distributed system, in which the exact runtime configuration varies. On the other hand, the implicit approach requires that the language's semantics define composite objects, synchronization, and communication boundaries. These boundaries are already clear in the explicit approach.

Process termination. Processes may be terminated explicitly or implicitly after a message has been processed. The difference in terms of implementation difficulties is minimal, but, in terms of runtime efficiency, the difference is significant.

Implicitly terminating processes after replying to a message is similar to a remote procedure call.[4] However, this approach results in execution inefficiencies because it means processes must be created and deleted in response to messages. The Actors model (and many Actors-based languages) is the best-known example of this approach: A process (actor) responds to a single message and then terminates. The Actors approach allows maximum concurrency but involves excessive process-creation overhead.

The alternative is to terminate processes explicitly. This approach lets processes continue after replying to messages and be available to respond to other messages. In this approach, fewer processes are created and deleted in response to messages. Abcl/1 operates this way.[5]

Process activation. Processes may activate when they are created or remain dormant until they receive a message. The first method causes more active parallelism because it lets processes run without messages, but it can waste resources. It's also difficult to implement, particularly in terms of resource management. POOL-T uses this more active approach.[6]

Most object-oriented languages wait for messages to prompt processes. As processes are spawned, the programmer must still manage synchronization and resource sharing. Also, this method creates more runtime overhead. Actors[2] and Concurrent Smalltalk[7] operate this way.

Process granularity. Granularity refers to the size of the schedulable unit of parallel activity and the amount of processing among messages. Programmers can vary the size of the processing unit to take advantage of hardware configurations and communication overhead. A language can support a combination of coarse-, medium-, or fine-grained units. Often, a language provides different synchronization mechanisms at different concurrency levels.

A finer granularity requires more efficient communication because dividing an object into smaller pieces increases communication. Coarser granularity is used when high communication overhead forces the programmer to compensate by increasing the size of a program's communicating units. Coarse granularity typically implies only one task per object or even per several objects; fine granularity implies multiple tasks per object.

With a coarse granularity, a program synchronizes only between objects because only one thread is active at a time. Other messages must wait for it to finish with each message. With a finer granularity, multiple threads have access to an object's variables, so the processor must synchronize its responses to these threads. Sometimes the processor spends more time synchronizing than computing.

The Actors model allows the finest granularity and the highest degree of concurrency. Abcl/1 supports medium granularity (lightweight tasks). Argus[8] supports coarse granularity (heavyweight tasks).

COMMUNICATION FEATURES

In conventional multiprocessors, objects communicate either by sharing memory or passing messages. But in object-oriented environments, communication is always through message-passing because sharing data among objects violates the encapsulation principle. Some argue that it is acceptable to violate the object paradigm at the physical level to maximize performance if it is accomplished by a verified compiler or runtime system.[1] However, the programmer should not have direct access to this physical level.

Table 2 summarizes the communication mechanisms of several concurrent object-oriented languages.

Message types. Object-oriented languages use three types of communication: synchronous, asynchronous, and eager invocation.

Synchronous communication uses remote procedure calls. It is easiest to implement, but sometimes wastes time because of the requirement for both the sender and receiver to rendezvous. Synchronous systems are more predictable and so are easier to verify. POOL-T's developers chose synchronous operation, believing that asynchronous communication causes unnecessary complications and carries the risk that things could get out of hand.

Some argue that it is acceptable to violate the object paradigm at the physical level to maximize performance if it is accomplished by a verified compiler or runtime system.

Asynchronous communication eliminates the wait for synchronization and can increase concurrent activity. But it is less predictable, hence harder to program and test. A program can use asynchronous communication if objects can keep processing without waiting for an answer to their messages. If objects need a reply, this must be explicitly programmed.

Eager invocation, or the futures method, is a variation of asynchronous communication. As in an asynchronous operation, the sender continues executing, but a future variable holds a place for the results. The sender processes until it accesses the future variable. If the results have been returned, the sender continues; if not, it blocks and waits for the results. Futures decrease or eliminate the wait for a reply and increase concurrency at a smaller risk to system consistency, but they add runtime overhead.

Messages in object-oriented languages specify the receiver's address. Abcl/1 supports two ways to specify the receiver: Its Parallel construct lets you send different messages to a group of receivers simultaneously. Its multicasting feature allows the asynchronous transmission of a message to a group of receiving objects. Both capabilities increase concurrency.

Message acceptance. Objects receive messages either implicitly or explicitly. Implicit acceptance means the system accepts messages automatically, and users cannot control the receivers. In an implicit system, a low-priority task can interrupt a high-priority task. To prevent this, the programmer can assign messages priorities, but this makes the program more complex.

Explicit acceptance lets objects control when they receive and process messages. This is more flexible because priority schemes are inherently defined in the list of messages an object can respond to. However, it increases runtime overhead because the system must prioritize the message queue, and consequently the programmer must assume more responsibility for controlling message processing.

Message processing and queues. Objects can process messages in the order received or in the order of the priority assigned to them by the system. Order-preserving queues are easier to design and test, but more difficult to implement. A system prioritizes messages by providing multiple queues of varying priority.

Languages that can prioritize messages

TABLE 2
COMMUNICATON FEATURES

Language	Message types S	A	F	Acceptance	Arrival	Queue	Synchronization
Abcl/1	X	X	X	Implicit	Deterministic	Priority	Critical sections
Abcl/R	X	X		Implicit	Deterministic	Priority	Critical sections
Actor	X	X	X	Implicit	Nondeterministic	Single	Interface
Argus	X	X	X	Explicit	Nondeterministic	Single	Critical sections
Concurrent Smalltalk	X	X	X	Implicit	Deterministic	Priority	Critical sections
COOL	X	X	X	N/A	N/A	N/A	Critical sections
Eiffel	X	X	X	Explicit	Deterministic	Single	Central
Emerald		X		Implicit	Deterministic	Single	Central
Gnu C++	X	X	X	Implicit	Deterministic	Single	Central
Hybrid	X			Implicit	Deterministic	Delay	Decentralized
Nexus	X	X	X	Explicit	Deterministic	Single	Interface
Parmacs	X	X	X	Explicit	Deterministic	Single	Critical sections
POOL-T	X			Implicit	Deterministic	Single	Central
Presto	X	X		N/A	N/A	N/A	Critical sections

S = Synchornous; A = Asynchronous; F = Futures

according to performance requirements are more flexible.[2] The underlying premise is that you cannot predict message arrival in a real communication network, so the architecture should be free to dynamically reconfigure itself to meet performance requirements. Although such prioritizing requires only one queue, its nondeterministic nature presents problems in implementing conceptually simple operations like terminating processes. You must also consider problems like limited queue size, missing messages, and messages that arrive out of order.

Synchronization. Correct synchronization coordinates parallel activities so that they run efficiently, consistently, and predictably. Too much coordination reduces concurrency; too little leads to undesired nondeterminism.

Inheritance complicates synchronization. When a subclass inherits from a base class, programs must sometimes redefine the synchronization constraints of the inherited method. This is the single most difficult aspect of integrating concurrency into object-oriented languages.

If a single centralized class explicitly controls message reception, all subclasses must rewrite this part each time a new method is added to the class. The subclass cannot simply inherit the synchronization code because the highest level class cannot invoke the new method.

The designers of POOL-T and Parallel Eiffel are faced with exactly this problem. The body of the object (called the live method in Extended Eiffel) specifies the concurrency constraints and must be rewritten each time a subclass with a new method is added. Languages with constructs like Select and Guards have centralized synchronization definitions. The Select construct allows the receiver to wait on several messages. It operates like a priority queue, in which priority is given to the first arriving message.

Critical sections are an alternative to a centralized synchronization. Critical sections can be used in each of an object's methods to maintain consistency; each method becomes responsible for controlling entry into the critical section. Locks, monitors, semaphores, mutual-exclusion mechanisms, and atomic variables can be used to control access to critical sections. The risk is that a subclass can modify the mutex. Inheritance makes it impossible for the system to guarantee that all subclasses follow the protocol for entering the critical section.

Hybrid[9] uses decentralized synchronization: Each method has a delay queue and explicit code to control it. Messages execute only if the delay queue is open. Changes in a superclass may be necessary if a subclass with a new delay queue is added. If methods in the superclass must control the new delay queue, the superclass must also be modified.

Actors-like languages, which receive messages implicitly, synchronize differently. Objects use a Becomes construct to specify replacement behavior, which also indicates what type of message to accept in the new behavior. The mail system delivers a message only when the object is receptive to it. All other messages are blocked. This interface approach has its own inheritance-mechanism difficulties. When new methods are added to a subclass, its existing methods may need modification to accommodate the Becomes construct.

Object managers can also control access by selecting authorized methods and blocking unauthorized ones. The method associated with the authorized message is executed and the next set of authorized methods are enabled when the current method executes a Becomes operation. The Becomes operation then enables the specified methods for execution. Act++ lets users name methods for each object state in the new behaviors.[10]

A similar approach in Rosette[11] uses enabled sets to define messages that are allowed in the object's next state. Objects pass the enabled sets' specifications from one state to the next. Enabled sets are themselves objects, and invoking their methods combines them. In this way, they can be built from inherited parts and locally extended. (Act++ does not support such a composition.) This eliminates monolithic code that must be rewritten to incorporate new synchronization constraints, but increases system complexity.

Guide[12] uses activation conditions to specify an object's state for executing a method. Activation conditions are complex expressions based on

- the number of messages received, completed, or executing;
- the state variables of the receiver; or
- the message contents.

Activation conditions are more expressive than enabled sets, but more difficult to use because the programmer must identify all states for which the method is enabled.

DELEGATION AND INHERITANCE

A language can extend an object's usability through delegation or inheritance, which strive to distribute knowledge throughout the system but are thwarted by atomic access and synchronization concerns. Table 3 compares how some concurrent object-oriented languages support inheritance.

Type. Delegation is based on the idea of subcontracting a job when the current object cannot perform it. If an object cannot answer a message, the sender sends a new message to an object that can. Delegation supports dynamic code sharing, which better accommodates growth because it lets you add messages without having to modify other objects extensively. The Actors model and Abcl/1 use delegation.

Inheritance lets you form specialized objects by inheriting methods and variables from a class hierarchy, which facilitates code reuse among classes. Subclasses behave as specialized versions of their parent.

Instantiation. Inheritance approaches vary from fully static to fully dynamic. In static inheritance, the compiler copies the inherited code. This eases the programmer's job because it supports reuse. In dynamic inheritance, which is more flexible but slows execution, the runtime system determines the appropriate active method and generates the correct execution thread. Dynamic inheritance is especially useful when method sharing is required.[4] In a concurrent environment, dynamic inheritance presents many problems.

The problems involved in implementing multiple inheritance in a concurrent object-oriented environment are like those encountered with single inheritance, but are compounded by the potential for conflicts among inherited methods and synchronization requirements. We know of no concurrent object-oriented language that supports multiple inheritance.

The easiest way to avoid inheritance problems is to leave inheritance out, as POOL-T's developers did. However, disallowing inheritance violates two of the seven characteristics of object-oriented languages and severely restricts the language's usefulness. (A new version of POOL-T with limited inheritance is being developed.)

Static inheritance copies the variable and method dictionaries into the inheriting object at compile time. This is simple and efficient, but it wastes memory by replicating code.

Another approach is on-demand inheritance, in which method dictionaries not explicitly defined in the class are located in the hierarchy and linked dynamically at runtime. However, this approach, used in Matroshka,[13] supports only static inheritance of the variable dictionary, so multiple copies of code still take up a lot of memory.

Another way to reduce the amount of memory static inheritance requires is to globally declare a primitive set of methods that all objects recognize. When these methods are invoked, the standard inheritance scheme is circumvented, so the default methods need not be copied into all objects. This approach requires version-control mechanisms to ensure that the latest version of the global method set is used.

In the recipe-query method, the receiving object asks another object (a proxy) for the method's recipe. After the proxy returns the recipe, the receiver can respond to the original message. This solution is simple, dynamic, and natural in a message-passing environment, but it has problems. First, the object requesting the recipe may need to access variables in the proxy, which can cause deadlocks if the proxy is locked. Second, in applications with a lot of object interaction, communications bandwidth is a concern. Third, it delays message processing until a recipe is received. Finally, it is difficult to update methods: The proxy that supplies a recipe must send messages to invalidate old recipes, or a version-control mechanism must ensure that the latest recipe is provided. (Flavors uses an automatic-update mechanism to control versions.[14])

A variation on the recipe-query approach is to regard the method as an object and have the receiver return the address of the method to the sender. The sender then sends a second message to the method. But this approach can also cause deadlock if the method needs information from the locked sending object. The alternative of sending the entire originating object's environment can also significantly increase communication traffic.[4]

We know of no concurrent object-oriented language that supports multiple inheritance.

Another method, computational reflection, lets a system access and manipulate a causally connected representation of its state. Any changes made to the system's self-representation are immediately reflected in its actual state and behavior. Each object has its own metaobject, which contains information about the implementation and interpretation of the object's methods, state, message queues, and evaluation method. Reflective computations modify the system's behavior. The system can modify itself in response

TABLE 3
INHERITANCE HANDLING

Language	Type	Classes	Instantiation
Abcl/1	Delegation	N/A	Dynamic
Abcl/R	Reflection	Single	Dynamic
Actor	Delegation	N/A	Dynamic
Argus	Copy	Single	Static
Concurrent Smalltalk	None	N/A	N/A
COOL	None	N/A	N/A
Eiffel	Copy	Single	Static
Emerald	None	N/A	N/A
Gnu C++	Copy	Single	Dynamic
Hybrid	Copy	Multiple	Static
Nexus	None	N/A	N/A
Parmacs	None	N/A	N/A
POOL-T	None	N/A	N/A
Presto	None	N/A	N/A

to changing conditions.

An example of a reflective system is a compiler-compiler that understands a language's structure. Patti Maes pioneered reflection in object-oriented languages by modifying KRS to accommodate reflection.[15] Since then, at least two others have used it: Rosette and Abcl/R[16] (a derivative of Abcl/1).

LANGUAGE SURVEY

None of the object-based languages we surveyed met Meyers' seven requirements because they either restrict or disallow inheritance. We based the selection of the languages for our survey on the availability of published material and detailed manuals. We also wrote a technical report that includes programming examples, which you can get by writing to us. Of course, this is not an exhaustive survey, but it highlights the issues, features, and trade-offs faced by language designers and implementers.

Some concurrent object-oriented languages are extensions to existing languages, like Smalltalk and C++. Of the many languages that add concurrency to Smalltalk, our survey examines Concurrent Smalltalk, which is the basis for a distributed version of the language. Several languages have extended C++. They range in approach from those that extend the syntax (COOL)[17], to those that provide predefined object types (Presto) or macros (Parmacs).[18] Many C++ extensions are designed to operate in a shared-memory environment; Gnu C++[19] is designed for a distributed environment.

Other languages have been developed with concurrency in mind, including Abcl/1, Argus, and Parallel Eiffel. The Actors model is the basis for many of these languages, including Abcl/1. Argus and Emerald[20] are supported by their own distributed object-oriented operating system; Emerald is not only a language but an operating system.

Which language you choose will depend on its conformance with the language, operating system, or hardware you use.

Actors. Developed at the Massachusetts Institute of Technology, Actors[2] is not an object-oriented language, but we include it and languages based on it because they support the highest degree of concurrency.

The basic unit in the language model is an actor, which consists of a mail address and a behavior. Actors may be created dynamically, and each actor has its own set of semantics. The language model supports neither explicit locks nor data sharing.

Each mail address is associated with an incoming-message queue. When a message arrives, the actor executes a script. The script accepts the message if it recognizes it; otherwise it rejects it.

The script may send messages to acquaintances (mail addresses known to the actor), to itself (usually by creating a copy of itself), or to an actor created specifically to handle the message.

The script also specifies a replacement behavior using the Becomes primitive. The replacement behavior is a new actor (with the same mailbox name) that accepts and acts on the next message. The actor enhances concurrency when it specifies the replacement behavior before responding to the newly received message. Expression actors handle futures by treating the expression (or future variable) as an actor and sending a message to evaluate it. The sender uses the mail address of the actor responsible for evaluating the expression as a placeholder for the actual value.

The actor delegates rejected messages to a proxy (another actor whose mail address is known to the sender). The proxy usually contains additional information (including an exception-handling mechanism) to respond to the message.

Continuation actors perform synchronous function calls by blocking until the synchronization event occurs. Actor-based languages synchronize with shared actors known as serializers, which protect their internal state against timing errors.

Abcl/1. Abcl/1 (An Object-Based Concurrent Language)[5] derives from Actors. Objects execute scripts that specify their behavior, acceptable messages, and responses. Independent objects may execute in parallel, but within an object, messages process serially. Objects are dormant when created and block until activated by a message. The object can select messages out of order by comparing them with a script pattern. It places messages that don't match the script at the end of a queue for processing later. After performing the actions, the object returns to dormancy by executing a Select construct.

Normally, message passing is asynchronous, point to point, and order preserving. A message can consist of tags to distinguish message type, parameters, and names of the sender and the reply destination. If a message is sent to an object that satisfies more than one message-pattern-constraint pair, the first pair specified in the script is executed.

Objects have two message queues: ordinary and express. Express messages can preempt ordinary messages, but not other express messages, atomic actions, or access to local persistent memory. After processing the express message, the object may abandon the preempted task, if so indicated by the express message. The object then takes the next message from the ordinary queue.

Abcl/1 has three types of message passing: past, now, and future. The past and future modes operate asynchronously; the now mode operates synchronously. Users specify parallel activity by using the Parallel construct within the object. Abcl/1 can also multicast — send the same message to a list of objects simultaneously.

Abd/R. A reflective version of Abcl/1, this language represents each object by a causally connected metaobject.[16] In turn, each metaobject may be represented by a meta-metaobject, creating an infinite tower of objects — although this is not normally done.

The metaobject represents an object's structural aspects and consists of state variables representing the methods to be executed, the state memory, a serial evaluator, and a message queue. Messages are sent to metaobjects when a computation involving the object is to be performed. The metaobject directs the object to perform the operation. The state variables perform the computation.

In other words, the metaobject is a generic template for an object implementation. Because the metaobject contains an object's internal structure, it can manipulate the structure by sending messages that cause the deletion, addition, or inheritance of methods.

Argus. Argus uses a transaction model and supports reliable computing.[8] It is both a language and an operating system. The language is an extension of Clu. Guardians encapsulate dynamic collections of objects and processes and provide the notion of a physical machine. They can be accessed by other guardians (and their internal variables manipulated) only through a procedure called a handler. A guardian is located at a single node in the network; several guardians may reside at a single node. They are designed to survive failures at that node by changing their node of residence. Guardians can create other guardians dynamically. A guardian's location is specified by the creator guardian.

Computations run as atomic transactions or actions. Actions are serializable and total. Totality implies that the action either completes successfully or is guaranteed to have no effect. Actions can be nested. Subactions may run concurrently, but a parent and a child action may not run concurrently.

Atomic objects provide synchronization. Each guardian in the system is assigned a unique identifier, as are threads within a guardian.

COOL. Developed at Stanford University, COOL (Concurrent Object-Oriented Language)[17] extends C++ to enable programming with medium- to large-grained concurrency in a shared-memory environment. Programmers can define parallelism within an object (each method can execute asynchronously), between objects (different methods in different objects can execute concurrently), and within a method (different functions within a method can be invoked in parallel).

By invoking methods defined as parallel, the execution can proceed asynchronously. Methods may also be declared as mutex, allowing synchronization at the function level on an object. Declaring a method as mutex allows multiple-reader, single-writer access to that object's public fields. Synchronization is against other public methods; private methods can be invoked from within the executing mutex method without locking. A release statement allows a mutex method to wait on an event while releasing the event to other methods. Mutex access to an object is more flexible and has more concurrency potential than monitors.

COOL provides fine-grained synchronization with future variables, spin locks, and blocking locks. Future variables add concurrency among methods. Like POOL-T and Presto, COOL does not support inheritance, friend or virtual functions, or method overloading.

None of the object-oriented languages we surveyed met Meyers' seven requirements because they either restrict or disallow inheritance.

Concurrent Smalltalk. The concurrent constructs added to Smalltalk-80 let the receiver continue executing after it receives a message (like POOL-T's postprocessing section).[7] Method calls are asynchronous; the sender may continue executing after sending a message. Asynchronous method calls are implemented using synchronous calls and an intermediary object called a CBox that blocks on behalf of the sender. The reply is returned via the CBox — the sender must send a message to the CBox to receive the reply.

In addition to asynchronous calls, Concurrent Smalltalk allows standard Smalltalk-80 synchronous calls. Synchronization is accomplished using atomic objects, which ensure that messages are accepted and executed serially. The language unifies objects and processes by defining concurrent object classes.[7]

Eiffel. Developed by the French Centre National de la Recherche Scientifique and Interactive Software Engineering, Eiffel[1] is capable of generic classes and assertions, as is C++. Generic classes allow for easy development of libraries for similar classes. C++ container classes and parameterized types make this possible. Assertions are simple formal specifications written as preconditions, postconditions, and invariants for method execution.

The parallel version of Eiffel has two synchronization mechanisms. The Wait-by-necessity mechanism is like a future, but does not require that you explicitly assign or invoke a future. This is syntactically cleaner, but does not have the flexibility of aggregate futures and future passing. The Serve mechanism implements an Ada-type rendezvous both semantically and syntactically.

Emerald. Developed at the University of Washington, Emerald's[20] primary goal is to demonstrate object mobility in a distributed system. A small set of language primitives locate and move objects, which the compiler implements as global, local, or direct. Remote invocations involve locating the object's unique global identifier in a hashed access table. There is no kernel intervention when objects within a single node are accessed.

Objects in Emerald are active if they are defined with a process. Otherwise, they are treated as passive data structures. Active objects invoke other objects, which in turn may invoke other active objects. An execution thread is initiated when active objects are created. Emerald permits threads to span multiple objects (either local or remote), as well as multiple threads contained within a single object.

Monitors are used for synchronization. Identically implemented objects on a single node can share code. A concrete object stores code for other objects. The code is dynamically linked and the object makes copies for remote nodes, if necessary.

Emerald provides call-by-move and call-by-visit method calls. In both, the argument objects are packaged along with the invocation message. In call-by-move, the object indicated is kept as an argument on the invoked node. Call-by-visit migrates the object to the invoking node.

ES-Kit C++. Developed at the Microelectronics and Computer Technology Corp., this extension of Gnu C++ for a distributed environment achieves parallelism primarily through futures.[19]

Aggregate futures support multiple invocations: The results can be collected in order of either arrival or invocation. Methods simulate barriers: A process can choose to wait for the completion of all invoked methods or for any method to complete.

An object represents a single thread of execution. Objects are uniquely identified with handles, which specify the node, application, class, and instance identification. A method's functions are addressed through pointers to the handles of the object that contains them.

Communication in the distributed environment is transparent to the user (C++ message-passing is the only communication visible to the programmer). This is achieved by creating a remote class that is overloaded with the actual method call.

Even languages that pemit inheritance support only single-class inheritance.

Hybrid. In Hybrid,[9] the unit of concurrency is a domain. A domain contains one or more objects, but processes them one at a time. Domains may be idle, active, or blocked. Idle domains may receive messages. Active domains may process requests only from the current thread of control, called an activity. Domains are blocked when remote procedure calls to objects in other domains are made.

An activity may be started by sending a message to a method called Reflex. Similar to forking a process, the parent and child activities can continue independently from this point. A delay queue can control access to one or more methods. An open queue allows the method to execute; a closed queue blocks access. Delay queues can also allow actions to be performed after a call has been returned.

Asynchronous met-hod calls with a return are executed by the Delegate construct. Execution resumes at the point of the method call when the sending object becomes idle again. Return values can be forwarded to the original caller, rather than to an intermediary. The atomic statement ensures mutual exclusion for a sequence of statements instead of a single method call. The Coloop and Coblock constructs allow a parent domain to block until a set of subactivities are performed.

Nexus. Developed at the University of Minnesota, Nexus[21] features a fault-tolerant, object-oriented, distributed operating system. This C-based language provides stable storage and atomic operations for transactions processing. Like Argus, it supports network transparency for inter-object communication.

Nexus is based on a concept of weak atomicity; the atomicity of an action is guaranteed only for changes made to that object, not to the actions performed on external objects as part of the sequence. That is, unlike Argus, it does not enforce strict serializability of concurrent actions.

A function library implements atomic transactions. In its model of object management, each object is identified with an object manager and all object managers for a class are grouped into a class manager. The class manager is implemented by a set of cooperating processes known as class representatives. Each representative maintains a list of the objects in that class associated with the controlling representative. The Nexus kernel interfaces only with the class representatives.

Parmacs. This language is essentially a set of C++ macros to facilitate the definition of concurrency and synchronization in a shared-memory environment.[18]

Parmacs organizes primitives in a hierarchy, with spin locks at the lowest level and monitors and barriers at higher levels. It provides two specialized types of monitors, getsub and askfor. Getsub is an atomic subscript server for loop-level parallelism. Askfor coordinates processes in a work queue and provides methods for defining queues. Barriers include leaklast (which allows the last process out of the barrier) and leakone (which allows one process out of the barrier).

POOL-T. Objects in POOL-T (Parallel Object-Oriented Language-T)[6] are active when they are created. The body of an object specifies an autonomous activity, and associated with each object is a specification and an implementation. The implementation unit contains the object's class definitions. The specification unit describes the classes and methods accessible by other objects. A root unit, consisting of class definitions, serves as the main program and execution starting point.

On startup, POOL-T implicitly creates an instance of the root unit's last class, which then must dynamically create more objects to execute. POOL-T follows Smalltalk-80's purist view of objects in that it considers every data item to be an object. This simplifies implementation but slows execution.

Message passing is synchronous and point-to-point. Messages are accepted explicitly, so objects cannot send and accept a message simultaneously. Messages are stored in one queue in order of arrival. When an object executes an answer statement, it answers the first message in the queue, whose name appears within the answer. This approach makes dequeueing more difficult. The Answer statement is similar to Ada's Select. Once the reply is sent, the receiver may execute a postprocessing section.

Presto. Like several C++ extensions, Presto,[3] also developed at the University of Washington, gives programmers a set of predefined C++ classes (such as threads and synchronization objects) to simplify parallel-program creation. Presto and its system of predefined threads and communication objects is designed for a global, shared-memory environment.

Thread objects are defined as a program counter plus a stack of invocation records. Threads can be created and activated dynamically, and they can be joined. Although threads can execute only one object at a time, an object can be assigned to multiple threads.

Presto's developers minimized the cost of creating threads by reusing threads from a reclaim pool. A scheduler object (created by the runtime system) schedules the threads as they are activated or resumed. Each physical processor is represented by a processor object, which requests executable threads from the scheduler. Scheduler objects cannot be migrated. Migration of objects can take place only when they are blocked (and resumed on a different processor).

A sending object may invoke an operation synchronously or asynchronously. The object's implementer decides whether the object executes sequentially or concurrently. Objects cannot determine if they are being invoked synchronously or asynchronously, and the invoking object cannot determine if the invocation is performed sequentially or in parallel.

Presto provides spin locks and blocking locks, as well as monitors and condition variables. Atomic integer variables provide fine-grained synchronization. Presto does not support method overloading, type checking of the argument list, or return values for method invocations.

Inheritance is the most difficult of Meyers' seven properties to implement in a parallel environment. Even the languages that permit inheritance support only single-class inheritance. Most of these were developed by an academic or research organization and have yet to reach widespread use.

Unless concurrency, synchronization, and communication are carefully integrated into a language, it can be inefficient or difficult to use. Some of the existing object-oriented languages can more easily be extended for parallel environments than others.

We need more research into issues of granularity of concurrent activity, synchronization mechanisms, and communication models. ♦

Barbara B. Wyatt is an engineering specialist at General Dynamics, where she is developing an object-oriented approach to software development in a multiprocessor environment. Her research interests include object-oriented design, concurrent programming, operating systems, and networks.

Wyatt received a BS in electrical engineering from Virginia Polytechnic Institute and an MS in computer-science engineering from the University of Texas at Arlington. She is a member of the IEEE Computer Society and ACM.

Krishna Kavi is a professor of computer science and engineering at the University of Texas at Arlington. His research interests are computer architectures, performance and reliability analysis, formal specification, program verification, and real-time systems. He is a member of the IEEE Computer Society Press editorial board.

Kavi received a BS in electrical engineering from the Indian Institute of Science and an MS and a PhD in computer science from Southern Methodist University. He is a senior member of the IEEE and a member of the ACM.

Steve Hufnagel is an assistant professor of computer science and engineering at the University of Texas at Arlington. His research interests include object-oriented software design, real-time systems, and distributed-processing systems.

Hufnagel received a BA in psychology and mathematics and a PhD in computer science from the University of Texas at Austin. He is a member of the IEEE Computer Society and ACM.

Address questions about this article to Wyatt at the CS and Eng. Dept., PO Box 19015, University of Texas, Arlington, TX 76019-0015; Internet wyatt@csr.uta.edu.

ACKNOWLEGMENTS

This work was supported in part by grant 3656-057 from the State of Texas Coordinating Board on Higher Education, Advanced Technology Program. We thank the ES-kit group at MCC for helping locate some reference material for this survey.

REFERENCES

1. B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, N.J., 1988.
2. G. Agha and C. Hewitt, "Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming," *Research Directions in Object-Oriented Programming*, B.D. Shriver and P. Wegner, eds., MIT Press, Cambridge, Mass., 1987, pp. 199-220.
3. B. Bershad, E. Lazawska, and H. Levy, "Presto: A System for Object-Oriented Parallel Programming," *Software — Practice and Experience*, Aug. 1988.
4. J.P. Briot and A Yonezawa, "Inheritance and Synchronization in Concurrent Object-Oriented Programming," *Proc. European Conf. Object-Oriented Programming*, Springer-Verlag, Berlin, 1987, pp. 32-40.
5. A. Yonezawa et al., "Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1," in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, eds., MIT Press, Cambridge, Mass., 1987, pp. 55-84.
6. P. America, "POOL-T: A Parallel Object-Oriented Language," in *Research Directions in Object-Oriented Programming*, B.D. Shriver and P. Wegner, eds., MIT Press, Cambridge, Mass., 1987, pp. 199-220.
7. Y. Yokote and M. Tokoro, "Experience and Evolution of Concurrent Smalltalk," *Proc. Conf. Object-Oriented Programming Languages, Systems, and Applications*, ACM Press, New York, 1987, pp. 406-415.
8. B. Liskov et al., "Implementation of Argus," *Proc. 11th Symp. Operating Systems Principles*, ACM Press, New York, 1987, pp. 111-122.
9. O. Nierstrasz, "Active Objects in Hybrid," *Proc. Conf. Object-Oriented Programming, Languages, Applications*, ACM Press, New York, 1987, pp. 243-253.
10. D. Kafura and K. Lee, "Inheritance in Actor-Based Concurrent Object-Oriented Languages," Tech. Report 88-53, CS Dept., Virginia Polytechnic Institute, Blacksburg, Va., 1988.
11. C. Tomlinson and V. Singh, "Inheritance and Synchronization with Enabled Sets," *Proc. Conf. Object-Oriented Programming Languages, Systems, and Applications*, ACM Press, New York, 1989, pp. 103-112.
12. Krakowiak, "Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications" *J. Object-Oriented Programming*, Sept./Oct. 1990, pp. 11-14.
13. L. Crowl, "A Uniform Object Model for Parallel Programming," *SIGPlan Notices*, April 1989, pp. 25-27.
14. D. Moon, "Object-Oriented Programming with Flavors," *Proc. Conf. Object-Oriented Programming, Languages, Applications*, ACM Press, New York, 1986.
15. P. Maes, "Concepts and Experiments in Computational Reflection," *SIGPlan Notices*, Oct. 1987, pp. 147-155.
16. T. Watanabe and A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language," *SIGPlan Notices*, Sept. 1988, pp. 306-315.
17. R. Chandra, A. Gupta, and J. Hennessy, "COOL: A Language for Parallel Programming," *Proc. 2nd Workshop on Programming Languages and Compilers for Parallel Computing*, IEEE CS Press, Los Alamitos, Calif., 1989.
18. B. Beck, "Shared-Memory Parallel Programming in C++," *IEEE Software*, July 1990, pp. 38-48.
19. K. Smith and A. Chatterjee, *A C++ Environment for Distributed Application Execution*, Tech. Report ACT-ESP-275-90, Microelectronics Computer Techonology Corp., Austin, Tex., 1990.
20. E. Jul et al., "Fine-Grained Mobility in the Emerald System," *ACM Trans. Computer System*, Feb. 1988, pp. 109-133.
21. A Tripathi, A. Ghonami, and J. Schmitz, "Object Management in the Nexus Distributed Operating System," Proc. Compcon, IEEE CS Press, Los Alamitos, Calif., 1987, pp. 50-53.