ABC++: Concurrency by inheritance in C++

by E. Arjomandi
W. O'Farrell
I. Kalas
G. Koblents
F. Ch. Eigler
G. R. Gao

Many attempts have been made to add concurrency to C++, often by extensive compiler extensions, but much of the work has not exploited the power of C++. This paper shows how the object-oriented facilities of C++ are powerful enough to encapsulate concurrency creation and control. We have developed a concurrent C++-based prototype system (ABC++) and describe how we can provide, with a standard compiler, almost all of the functionality offered by a new or extended language. Active objects, object distribution, selective method acceptance, and synchronous and asynchronous object interaction are supported. Concurrency control and synchronization are encapsulated at the active object level. The goal of ABC++ is to allow users to write concurrent programs without dealing with explicit synchronization and mutual exclusion constructs, with as few restrictions on the use of C++ as possible. ABC++ can be implemented on either a shared memory multiprocessor or a cluster of homogeneous workstations. It is presently implemented on a network of RISC System/6000® processors and on the IBM Scalable POWERparallel™ System 1 (SP1™).

The object-oriented programming (OOP) paradigm provides the tools and facilities for developing software that is easier to build, extend, reuse, modify, and maintain. The key concept in the OOP paradigm is the building of programs around *objects*, as opposed to around actions, as in the traditional procedural approach.[1] An object is a self-contained entity that has exclusive control over its own internal state, and communicates with other objects by sending them messages. The fact that OOP supports the building of software around encapsulated objects suggests that the OOP paradigm may present an ideal environment for concurrent programming.

Traditionally, users writing concurrent programs are concerned with threads of control and problems of synchronization and mutual exclusion. Writing, maintaining, debugging, extending, and reusing concurrent software is extremely difficult. However, with the advent of inexpensive multiprocessors and high-performance workstations, as well as fast and reliable communication networks, concurrent programming has become an integral part of language design.

The integration of concurrent programming and object-oriented programming attempts to bring the benefits of object-oriented programming to concurrent programming, to ease the task of writing concurrent programs. A common way to integrate these two paradigms is to encapsulate concurrency creation, synchronization, and mutual exclusion at the object level. Such an object is called *active*. The notions of object and process are unified into a single notion of an active object

0018-8670/95/$3.00 © 1995 IBM

that contains (possesses) its own thread of control. In contrast, a passive object does not have its own thread of control and has to rely on active objects to assume control or on other synchronization mechanisms (such as locks, monitors, etc.) to ensure its integrity. The thread of an active object executes a particular method of the active object named the "body." An active object synchronizes with other active objects by using accept statements or similar constructs that specify the set of methods that the active object is prepared to serve at a given time, depending on its internal state.

This paper describes the design and implementation of the prototype ABC++ (Active Base Class). We show that with a standard C++ compiler and no preprocessing, C++ is powerful enough to allow the integration of the paradigms of concurrency and OOP. The goal of ABC++ is to allow its users to write concurrent programs, using a standard C++ compiler, without having to deal with explicit synchronization and mutual exclusion constructs, and with as few restrictions on the use of C++ as possible.

Our initial attempt in designing and implementing ABC++ concentrated on proving that C++ is powerful enough to allow active object creation, object distribution, and a variety of active object interactions. Our first prototype implementation imposes a few restrictions on the use of the C++ language that will be discussed later.

ABC++ is presently implemented on a network of RISC System/6000* processors using Transmission Control Protocol/Internet Protocol (TCP/IP) for interprocessor communication, and on IBM's distributed memory multiprocessor SP1* using a high-performance message-passing library (EUIH). It is written in C++, with the exception of a few lines of code that are written in assembly language. All aspects of ABC++ described in this paper, unless stated otherwise, have been implemented and tested. We plan to migrate ABC++ to other platforms including a cluster of SUN SPARCStation** 10 computers.

This paper assumes some background in concurrent programming, object-oriented programming, and the C++ language. For more information on concurrent programming, the reader is directed to Reference 2. Meyer's book on object-oriented software construction[1] provides a good introduction to object-oriented programming, and a book by Stroustrup provides a good introduction to C++.[3]

Issues in object distribution and in creating active objects are first discussed. The details of how active objects are created and how objects interact in ABC++ are then presented. Concluding remarks are followed by an appendix that gives an example written in ABC++.

A standard C++ compiler and no preprocessing supports the integration of concurrency and OOP.

Concurrent systems

An important aspect of a concurrent system is its memory model. Most concurrent languages use either distributed memory or shared memory. In a distributed memory model, processors have exclusive access to their own local memories and communicate with other processors through messages. In a shared memory model, all processors share a large global memory and communication is through shared variables—that is, a processor writes into the shared variable, which can then be read by other processors. Of course, access to the shared variable must be synchronized.

Communication in a distributed memory model can be classified into two paradigms. In the first paradigm, objects and processes interact by sending and receiving messages. A send or a receive may be blocking or nonblocking. With blocking, the caller is blocked as soon as the call is issued, until a reply is received. The second paradigm of interaction consists of call and reply. This paradigm is an extension of the sequential procedure call, and is commonly referred to as a remote procedure call (RPC). In an RPC communication, objects request services of remote objects with familiar procedure call syntax. RPCs may also be blocking or nonblocking. In this paper we will not

discuss the different models of RPCs introduced in various operating systems (that is, "at most once," "at least once," etc., semantics). As in most other concurrent object-oriented languages, what we refer to as RPC has essentially the same syntax and semantics as the regular method invocation, where *method* is a term used in object-oriented languages for member functions.

Concurrent C++-based systems

C++[3] has increasingly become the language of choice among developers, and numerous attempts have been made to add concurrency to it.[4–12] For a more complete review of the literature and approaches taken in adding concurrency to C++, see Reference 13. Two approaches can be used to add concurrency to an object-oriented language such as C++. In the first approach, the language is extended in order to add the concurrency constructs. New or extended languages can use the compiler to provide higher-level constructs, compile-time type checking, and enhanced performance. In the second approach, such as that used with ABC++, the facilities of OOP are used to encapsulate the lower-level details of concurrency. In this second approach, a library class generally referred to as a Task class provides the concurrent facilities. A user wishing to write concurrent code can use Task, normally by inheriting from it. In this approach the concurrency constructs are kept outside of the language, the language is kept small, the programmer can work with familiar compilers and tools, the option of supporting many concurrent models through a variety of libraries is provided, and the porting of code to other architectures is eased (usually, a small amount of assembler code needs to be changed). Software developers typically have large investments in existing code and are reluctant to adopt a new language. A class library with sufficient flexibility that can provide most of the functionality of a new or extended language is often more palatable.

Concurrent C++-based systems often require extensive compiler extensions. Previous attempts to add concurrency to C++ without compiler extensions have imposed unreasonable limitations on the users.[4,8,14,15] These limitations can include using explicit message queues in object interaction, limiting the number of inheritance levels to one, explicitly managing threads through the use of start routines and managing synchronization and mutual exclusion through the use of explicit mechanisms to wait on an event and to signal events. In these systems, after an object has been created, in a separate step the object is activated by the use of a start routine, provided by the class library. These systems are primarily thread packages and have not attempted to exploit the object-oriented facilities of C++ for concurrency creation and control. Buhr and Ditchfield[16] argue that many of these problems are difficult or impossible to solve without compiler support. For this reason, Buhr et al.[5] employ language extensions to provide the users with sufficient flexibility without imposing significant limitations. In the remainder of this section we review some of the C++-based concurrent systems that, with the exception of ES-Kit, have not changed either the C++ language or the C++ compiler.

ABC++ supports implicit concurrency through active objects that possess their own thread of control.

AT&T. The AT&T Task Coroutine Library[14] is one of the earliest concurrent C++ libraries. A class wishing to use the concurrent facilities of AT&T's Task library would be derived from class Task. Objects of a class derived from Task have their own thread of control, and run within the same UNIX** process. AT&T's Task library imposes the following limitations on its users:

- Only a single level of derivation is allowed from Task, hence, no derivation is allowed from a user's class.
- Objects of a class deriving from Task communicate through explicit message queues.
- Objects synchronize through explicit use of wait and alert routines.

Doeppner. Doeppner's[8] Task library supports true concurrency by building on top of a thread package named Threads. However, it still suffers

from the same limitations as in AT&T's Task library.

PRESTO. PRESTO[4] is a library for programming with threads. Concurrent programming facilities are provided by a set of classes called Thread, Lock, Spinlock, Condition, and Monitors. Objects of a class wishing to use PRESTO are not given a thread. Threads are created independently by the C++ new operator. The Thread class provides a start method, which can be invoked on a thread object. The arguments to start include the object whose method is being invoked, the method name, and the method arguments. Class Monitor provides the methods entry and exit, which are explicitly used to ensure atomicity when accessing critical sections. Condition variables are instances of class Condition, and are manipulated by the methods signal and wait. Communication among PRESTO objects is solely through shared variables.

AWESIME. AWESIME[15] is very similar to PRESTO, with additional support for process-oriented simulation. Unlike the AT&T Task library, it allows arbitrary levels of subclassing from its Task class (called THREAD). However, thread and message queue management in AWESIME are still explicit.

Gautron. Gautron[17] extends AT&T's library by adding support for LIFO-mode (last in-first out) task scheduling, priorities, and user-controlled scheduling. The limitations of the AT&T class library are not addressed.

Amber. Amber[18] is a C++-based system for writing distributed applications on a homogeneous network of 64-bit multiprocessors. Amber's concurrency model is the same as that for PRESTO, where thread objects are created independently and manipulated by a start routine. Amber does not modify the language or the compiler; however, it requires preprocessing of the code before compilation by C++. It assumes that all nodes share a globally managed virtual address space. In this paper, the words "processor" and "node" are used interchangeably to refer to a single processing element. A shared virtual address space will quickly exhaust its virtual address space on machines with 32-bit addressing; thus Amber assumes the availability of 64-bit addressing machines. Amber objects are passive, and their methods can be invoked locally or remotely. The active objects of the system are thread objects that are migrated to a node in the network where an object whose method has been invoked resides. The preprocessor assists in trapping method invocation on remote objects. More detail on how Amber supports object distribution is provided in the section "Issues in object distribution."

PANDA. PANDA[19] is a run-time package that supports distributed applications in C++. PANDA does not extend the C++ compiler; however, a preprocessor is used. The preprocessor inserts source code at specified places in the user code. As in Amber, PANDA assumes that all nodes share a globally managed virtual address space, hence it requires the availability of 64-bit processors. Its distribution facilities are very similar to those of Amber.

ES-Kit. ES-Kit[7] is an object-oriented system for distributed applications. Developed at the MicroElectronics and Computer Corporation, this experimental system is written in GNU C++. ES-Kit objects are not active, and parallelism is created by method invocation of many objects. ES-Kit relies on some nonstandard features of GNU C++ for object distribution and remote object creation. In particular, ES-Kit relies on a nonstandard member access operator, →, which takes four arguments (the standard member access operator is a unary operator). Therefore, in effect ES-Kit has changed the compiler. It uses global identifiers for object location, and each object is assigned a *handle* at the time of its creation that contains the unique identifier for the object, including its node-id. The nonstandard features of ES-Kit assist in trapping method invocations and address translation at node boundaries.

ABC++. AT&T, Doeppner's, and AWESIME task libraries cannot be classified as concurrent object-oriented systems, since they have not encapsulated concurrency creation and control. PRESTO is a thread package that can be used in building concurrent object-oriented systems. In the section "Issues in creating active objects," we analyze why some of these task libraries have failed to encapsulate concurrency creation and control at the active object level.

ABC++ supports *implicit* concurrency through active objects that possess their own thread of control and can run simultaneously with other active objects on a shared or distributed memory

multiprocessor, as well as on a homogeneous cluster of workstations. Concurrency creation is implicit, meaning active objects are created like ordinary C++ objects without the use of explicit start routines for thread activation. Each active object has a mailbox and manages its own message queue without requiring the users of ABC++ to manipulate message queues as in References 8, 14, 15, and 17. An active object in ABC++ has exactly one thread of control and is able to process one message at a time, hence ABC++ programmers do not use explicit synchronization and mutual exclusion constructs to achieve atomicity.

ABC++ programmers do not use explicit synchronization and mutual exclusion constructs.

Active objects in ABC++ communicate via method invocation. An invocation of a method of a remote object is automatically transformed into a remote procedure call. Selective method acceptance and asynchronous communications are also supported.

The restrictions that ABC++ presently imposes on the use of the C++ language are:

- Public member functions of active objects must be virtual.
- Friend classes and friend functions are restricted to access virtual member functions only. In C++, an external function or another class may be defined as a friend of a class, allowing access to nonpublic members of the class.
- No public or static class data members are allowed.
- The types of arguments allowed for methods of active objects as well as return results are restricted to simple types and pointers to active objects. The number of such arguments is limited to seven.
- The number of public member functions is fixed by ABC++ to a relatively large number that can be modified (this number is 32 in the present implementation).

The first restriction can easily be eliminated by a simple preprocessor. The second and third restrictions cannot be eliminated unless we change the model of ABC++ and provide explicit mechanisms for achieving atomicity. We believe the fourth restriction can easily be eliminated by a more careful implementation of ABC++ and with the use of function templates. Work is presently under way to achieve this goal.

Issues in object distribution

In a distributed memory model, it is important to provide support for location-independent object interaction. This is particularly important when objects are allowed to migrate. There have been many efforts by researchers over the years to represent objects so that they may be referred to by remote processors. One solution to this problem is to use global identifiers.[7] Global identifiers are used in place of object references when objects are referenced. The disadvantage of this technique is that it requires extensive compiler support in translating these identifiers into local references each time they cross a node boundary. ES-Kit uses global identifiers, and a nonstandard member access operator is used to assist in address translation at node boundaries.

Another solution is to represent remote objects by *proxies*. Shapiro[20] introduced proxies as local representatives of remote objects. A proxy object contains some information about the object it is representing, including either the address of the object or the address of a name server where the object address may be obtained. Proxies assist in providing location transparency in a system where objects are distributed across many nodes. They allow uniform invocation of methods, irrespective of their location, and are commonly used to facilitate object interactions in a distributed memory environment.[21–23]

Amber uses proxies (called object descriptors in this case) and globally shared virtual address space to provide location transparency. The virtual address space on each node is identical. Program code and static data are replicated on all nodes at the same virtual addresses. Each node in the network has a distinct region of the address space for allocating new objects. No node in the

network would use the region of another node when it creates new objects. Although the solution for Amber avoids address translation at processor boundaries, it suffers from the problem that large amounts of virtual space must be "wasted" on objects that reside on other nodes. This virtual memory requirement can be a disadvantage in distributed memory parallel computers containing large numbers of simple processors, since they may not otherwise require virtual memory hardware. As was previously mentioned, a shared virtual global address space will soon exhaust its virtual address space on a 32-bit addressing machine and, for this reason, the Amber model of computation assumes the availability of 64-bit architectures.

Our ABC++ solution is similar to that of Amber, but it does not require virtual addressing or any special hardware.

Issues in creating active objects

In C++, objects "know" how to initialize themselves (through *constructors*) and how to clean up after themselves (through *destructors*) when they are no longer needed. Ideally, an active object should behave in a similar fashion. It should encapsulate thread and message queue management as well as object construction and destruction. However, as outlined in Reference 16, if the C++ compiler is not extended to empower active objects, many challenging problems are created.

Existing task libraries are largely based on the AT&T class library. These systems either fail to create active objects implicitly, or impose unreasonable restrictions on users (e.g., prohibiting inheritance). An abstract base class called Task is used as a parent class for any derived class where concurrency is desired. The creation of an active object in these systems causes a thread to be spun off that will execute the "body" of the active object. Two approaches are used in the selection of a body for the active object. The first approach uses the constructor of the derived class as the body. The second approach defines a "Main" routine as the body of the active object.

If the constructor of the derived class is used as the body, it must be written as an infinite loop; therefore it will not terminate. A nonterminating constructor prohibits inheritance from this class, as is the case in References 8 and 14. This is due to the fact that in C++, the ancestor classes perform their construction before descendant classes. Therefore if a constructor is nonterminating, no derived constructor would be able to complete its tasks, hence prohibiting inheritance.

We now consider the case where a Main is defined as the body of the active object. If such systems allow arbitrary level of inheritance, object construction is faced with significant problems. As was previously mentioned, the constructors of ancestor classes perform their duties before constructors of successor classes. Hence, the Task constructor performs its duties before any other constructor. If the Task constructor is responsible for spinning off a new thread, it must then direct one thread to the object that called new, and give the second thread to the new active object, which will execute Main. If the number of levels of inheritance is unknown, and if it is further unknown which levels define constructors and which do not, the Task constructor cannot know where to return the thread in the calling object. Furthermore, when the Task constructor is called, the new active object is not fully constructed, hence should not receive a thread. If the active object receives a thread at this point, we are faced with the problem of premature method invocation. C++ itself does not control when the creating object can invoke methods of the newly created object; thus such systems require the use of a start routine for explicit thread management, as in Reference 15.

An active object must not be destroyed until its thread of control has terminated. Once again, C++ itself cannot ensure thread termination before object destruction. If compiler extensions are not made, one way to solve this problem is through explicit use of a wait routine, as in References 8 and 14. Explicit use of start and wait routines is error-prone and complicates the already complex task of concurrent programming. One of the objectives of concurrent object-oriented languages is to reduce the use of explicit synchronization and mutual exclusion mechanisms, thereby reducing the complexity of writing concurrent programs. Similarly, active objects should encapsulate message queue management. Existing Task-based systems typically require explicit message queue management, as in References 8, 14, and 15.

So far our discussions have focused on creating active objects in a shared memory environment. A distributed memory model poses many different problems, as previously mentioned. The existing Task libraries only support a shared memory model. ABC++ supports implicit concurrency through active objects in a shared or a distributed memory model. In the next section, we explain how active objects are created in ABC++.

Location transparency eases the task of writing concurrent programs in a distributed memory model.

ABC++

In this section we outline how ABC++ supports active object creation, object distribution, selective method acceptance, and object interaction.

Object distribution. Since ABC++ supports object distribution, it is vital to provide location transparency for active objects. Location transparency eases the task of writing concurrent programs in a distributed memory model by freeing the programmer from having to deal with object location. We adopt the proxy approach to provide location transparency; however, we refer to proxies as *object descriptors.*

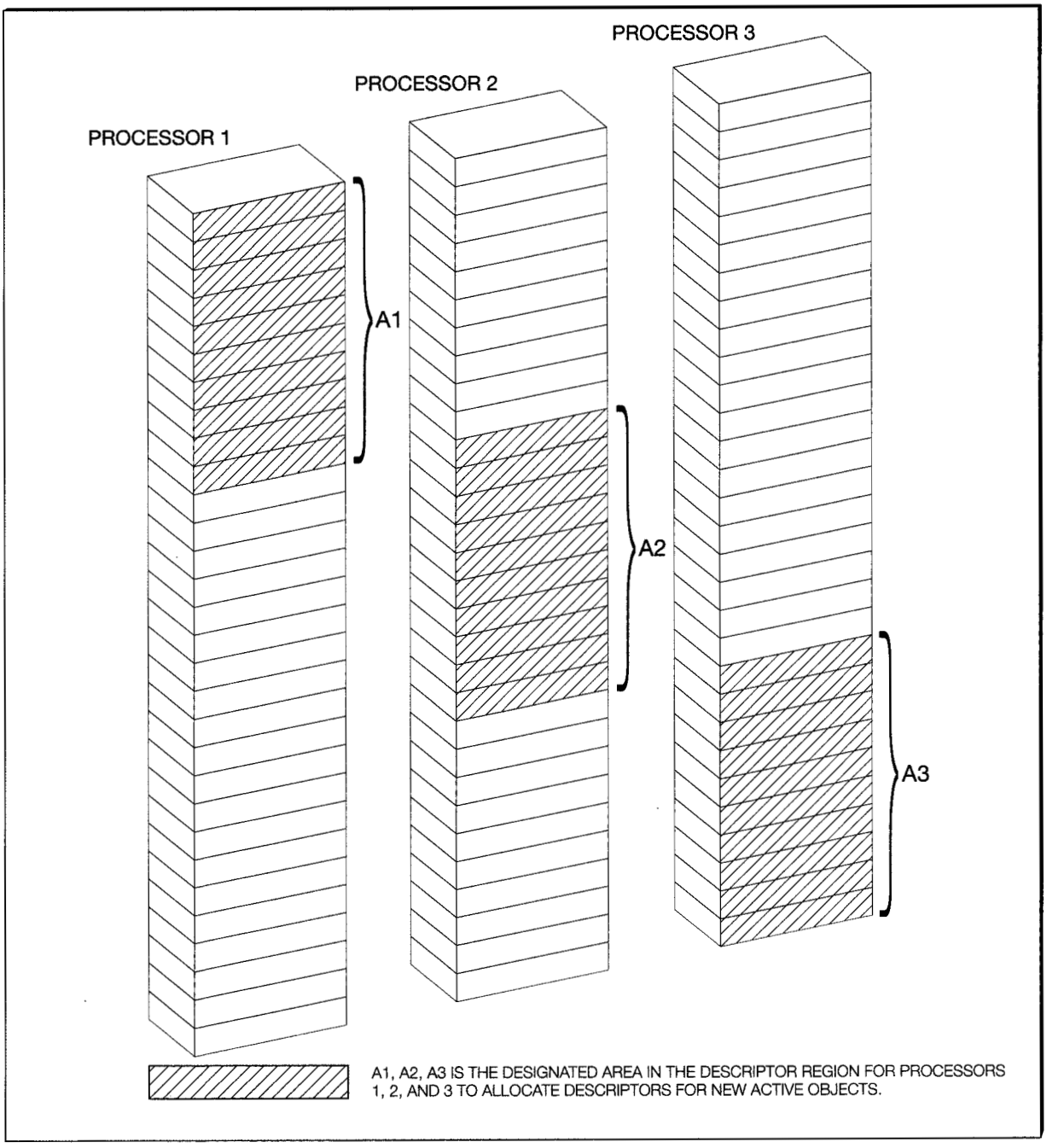
A designated area of memory, referred to as the *descriptor region*, is allocated on all the nodes at the same address. This region is essentially a table of object descriptors. An object descriptor will contain some information about the object it is representing. Each active object has an object descriptor representing it on each of the participating nodes at exactly the same location. The object descriptor on the node where the active object resides is called a local object descriptor (LOD). All other object descriptors are called remote object descriptors (RODs). The entries containing LODs also contain the memory address of the active object.

Since object descriptors are small in size (only a few words), we can define a relatively large descriptor region at startup time. ABC++ performs "garbage collection" (the reuse of unused or discarded items) on descriptors; therefore it is unlikely that processors would run out of descriptors. However, it is possible to repeat the process of creating object descriptor regions, if necessary. The communication layer of ABC++ establishes communication across node boundaries. This layer, among other things, provides a daemon thread on each of the participating nodes at startup time. A daemon thread is a system thread that is unknown to the user. The daemon threads handle various housekeeping tasks including polling for communication. When a processor runs out of space in its designated area for storing new object descriptors, a descriptor region creation can be initiated by sending the daemon thread a message (this feature is not implemented in our present ABC++ prototype implementation).

To ensure that processors can create objects freely without having to synchronize with other nodes, each processor in the pool is assigned a specific area of the descriptor region in which its new object descriptors are to be stored. No processor in the pool would use the designated region of another processor for storing new object descriptors. Figure 1 shows the descriptor regions in a three-processor cluster and their designated areas for storing new object descriptors. The object descriptor *index* determines the identity of the node that created the corresponding active object. If objects never move after their creation, the descriptor regions on all processors will always contain RODs everywhere except in their designated areas, which will contain LODs only.

In ABC++, when an active object is created, an LOD object is created (represented as an object) on the originating node and stored in the first available entry of the processor's designated area in the descriptor region. The address of the corresponding active object is also stored in that same entry. For example, in Figure 1, when a new active object is created on processor 1, an LOD is created and stored in the first available entry of the area marked as A1. ROD objects are created at startup time by ABC++.

Figure 1 Object descriptor region in a three-processor machine cluster



In C++, when an object is created, its address is returned to the program that initiated the object creation. In ABC++, the address of the descriptor region entry where the LOD is stored, is returned to the caller. Therefore, ABC++ programs deal with descriptor region addresses, not

active object addresses. We refer to the processor that created an object as the object's *home node*. While an object is being constructed on a particular node, no action takes place on any of the other nodes with respect to the newly created object. However, if a reference to this new object descriptor is passed to other nodes, the corresponding object descriptor already exists and can determine the home node of the corresponding object from the object descriptor index.

Hence, in ABC++ a reference to a descriptor for a local object can be quickly translated into the object's address, and that same reference can be sent to another processor and will automatically refer to the correct ROD on that node. Thus, descriptor region addresses can be freely passed as arguments to remote procedure calls, as return results from remote calls, or within messages of any kind. A descriptor region address always refers to a correct and current LOD or ROD, as appropriate.

ABC++ gains much of its power by exploiting the virtual function mechanism in C++. For this reason, it requires that all the methods on the interface of active objects (that is, *public* member functions) be declared as *virtual*. In C++, the visible members of a class are specified as public. Members declared as *private* are not externally visible. In C++, an object with virtual methods will contain the address of a table, commonly referred to as the *vtable*, which contains pointers to virtual methods. When a virtual method of an object is invoked, only its vtable index is resolved at compile time. At run time this index is used to access a vtable that would contain a pointer to the appropriate method.

As was mentioned earlier, ABC++ returns the descriptor region address where the LOD of a new active object resides. Hence, when a method of an active object is invoked, this invocation is intercepted by the LOD object and before the real method is invoked, ABC++ performs some tasks. This is achieved due to the fact that object descriptors are real C++ objects and contain a vtable pointer. ABC++ provides two classes, LOD_proto and ROD_proto, instances of which are LODs and RODs, respectively. These classes define a set of private virtual member functions. We refer to these virtual functions as *protocol functions*. Hence LOD and ROD objects will contain a vtable pointer. There is a one-to-one correspondence between the protocol functions and the virtual methods in the user's class hierarchy. Therefore, the maximum number of member functions in user classes is limited by the number of protocol functions (this number is 32 in the present implementation).

The interception of method invocation in active objects would work if the vtable pointer in the object descriptors and in the active objects is at the same location. Since object descriptors are small, the vtable pointer must also reside near the beginning of the object. Some compilers place the vtable pointer in the first word of the object, whereas others place the vtable pointer after the first nonvirtual base class having a virtual method. In C++ virtual base classes are used to avoid multiple copies of shared classes along multiple inheritance paths. To ensure that the vtable pointer for object descriptors and active objects is at the same location, LOD_proto, ROD_proto, and the base class for active objects have no data members. In the case of active objects, since user classes inherit from ABC++, it is sufficient to make the first base class of ABC++ an abstract base class with a single pure virtual member function main (LOD_proto and ROD_proto also define main in order to maintain the one-to-one correspondence between the methods of active objects and the protocol functions). In C++, abstract base classes are never instantiated. Their role is to provide a common interface that will be implemented by the derived classes. A pure virtual member function is designated by following its argument list with the keyword "=" and "0." The definition of such a function is provided by derived classes. An abstract base class must contain at least one pure virtual function. When a method of a local active object is invoked, the following steps take place:

- The corresponding protocol function of the LOD is invoked.
- The protocol function checks to see if the active object is presently serving messages of the type requested and if the object is free to serve this request. As was mentioned earlier, ABC++ supports selective method acceptance. Namely, active objects can change their interface depending on their state. A following section provides more detail on how selective method acceptance is implemented. If the object is serving such requests and is free, the appropriate method of the object whose local address is

stored in the same entry of the descriptor region as the LOD object is invoked.
- If the invoked method is not being served at this point, or if the object is busy serving other requests, the protocol function queues up this request for later processing. The protocol function blocks other functions until a reply is received.

When a method of a remote active object is invoked, once again this call is intercepted by the ROD object and the corresponding protocol function of the ROD object is invoked. The protocol functions of ROD objects forward the call to the node where the object resides.

In the present version of ABC++, we have not implemented object migration. However, the technique used by ABC++ for providing location transparency allows object migration. Recall that each processor has its own designated area for storing new object descriptors. Therefore if an object moves to a new node, the following steps would have to take place:

- An ROD object is instantiated and stored in place of the LOD object.
- A message is sent to its home node informing it of the new location.
- An LOD object is instantiated in the destination node and stored in place of the ROD. The local address of the active object is also stored in the same entry.

In this scheme, the home node of active objects contains information about their whereabouts as they move around. Therefore, to locate a remote object, its home node must be consulted. Once again, the daemon thread would assist in achieving object migration. In this technique, all remote accesses have an extra level of indirection. If the extra level of indirection is too expensive, other techniques such as broadcasting can be used to keep the LODs and the RODs up-to-date.

Creating active objects. ABC++ is able to create implicit concurrency (and provide many other features) by exploiting the virtual function mechanism of C++. This allows ABC++ to intercept the method invocation mechanism of C++. In the previous section, we explained that when active objects are created, the address of the object descriptor is returned to the caller; therefore all method invocations on active objects are intercepted, and before the "real method" is invoked, a corresponding protocol function of the object descriptor is invoked. In this section we address how active objects are created when they are allocated dynamically by operator new. ABC++ is able to create automatic (that is, stack-based) objects, the details of which are not addressed in this paper.

To use ABC++, user classes must inherit from Task. All classes that inherit from Task should declare the methods that would be invokable by other active objects as virtual. ABC++ provides a virtual main that becomes the body of the active objects. This default main will accept all messages in FIFO (first in-first out) order. Users of ABC++ may define their own main.

Functions that support active object creation. C++ allows programmers to redefine almost all C++ operators. This redefinition is called *overloading*. The creation of active objects in ABC++ is handled by two overloaded new operators, the Task constructor, and a static member function of Task, called *auto-start*. We next outline the duties of these functions.

Operator new is a request for creating an active object that might be local or remote. An optional argument to new provides the remote node identifier. The overloaded version of new for local object creation allocates storage, instantiates class LOD_proto, and initiates the chain of constructor calls before returning. If the request is for remote object creation, the overloaded version of new for remote object creation sends a message to the daemon thread on the requested node, which would arrange for a local object creation on the remote site. Object construction on the site that requested the remote object creation is aborted.

We previously outlined some of the problems in creating active objects and, in particular, we addressed the problem of returning the original thread (the thread that called new) to a proper statement in the calling object, and giving the new object a thread only after it is fully constructed. We refer to the address of the proper statement in the calling object as the *return address*. The necessary steps for recording where this information may be found are taken by new. It terminates by returning the address of the descriptor region entry containing the corresponding LOD object.

In the case of local object creation, after new returns, the chain of constructor calls begins until the Task constructor is called.

The Task constructor performs the following duties:

- It saves the return address to be used by the auto-start function. The overloaded new has recorded where this information may be found. This address is obtained by a small amount of assembly code. This is the only place in ABC++ that some assembly code is used.
- It replaces the return address with the address of auto-start.
- It saves the address of the newly created object in the same entry of the descriptor region where the corresponding LOD object is stored.

After the Task constructor terminates, the object construction begins, and eventually control is given to auto-start.

Since a pointer to auto-start is left by the Task constructor where the return address is normally found, after the most derived constructor (that is, the first constructor called after new returns) performs its duties, control is given to auto-start. Auto-start performs the following duties:

- It spins off a thread for the newly created object and makes that object's virtual method main its body.
- It returns the original thread to the object that called new.

Main defines the activity of the object. It also processes the incoming messages by issuing accept statements. Provided that users use virtual functions, ABC++ treats object descriptors as if they are regular C++ objects, with the virtual function mechanism translating the calls to local or remote invocations as appropriate. Within this paradigm, users would never need to know when an object is remote and when it is not.

Selective method acceptance. Active object systems must be able to selectively accept messages to answer, and to delay some for later acceptance. If a C++ concurrent system is to have this capability, then active objects should be able to control which of their methods are "invokable." C++ provides no native mechanism for this, and there has previously been no class-library-based approach to solving this problem. Some class libraries[8] require users to use explicit message queues, thus erasing the natural object-oriented analogy of methods to messages.

In ABC++ each active object has a mailbox and manages its own message queue. As was mentioned earlier, there is a daemon thread on each of the participating nodes. The daemon thread delivers the newly arrived messages to the appropriate mailboxes.

To solve the selective method acceptance, the Task class provides an accept method and the necessary data structures for implementing the required message acceptance protocol. The arguments to the accept method (beside the this pointer) are an integer and the appropriate virtual method function pointers. The integer argument represents the total number of virtual function pointers appearing in the accept statement. The virtual function pointers define the set of acceptable messages. The accept method flips the appropriate bits in a *method-mask* according to the value of its parameters. Method-mask is an instance variable of class Task, and the indicator bit is a "1" if the corresponding method is currently being accepted by the object; otherwise it is a "0." A method is "open" if it is currently acceptable, otherwise it is "closed." ABC++ users can change the interface of an active object by issuing accept statements from within the body of the active object. For example, the statement accept(1, push) causes push messages to be accepted by a stack object. The pop messages are delayed. ABC++ intercepts method invocations and an appropriate protocol function is called before the "real method" is invoked. The protocol functions invoke the "real method," if the method is currently acceptable by the active object; otherwise, the message remains queued for later processing.

Premature method invocation and object destruction. An active object in ABC++ is not given a thread of control until it is fully constructed. Therefore, premature method invocation is no longer an issue.

In ABC++, active objects are destroyed in the same way C++ objects are destroyed (for example, by delete, in the case of dynamic objects). To solve the problem of premature object destruction, once again we use the protocol function

mechanism. Task provides a virtual destructor. Therefore, the destructors of all user classes inheriting from Task will also be virtual. Since ABC++ intercepts invocations made to all virtual methods, the destructor calls are also intercepted. The destructor of an active object is not called until its thread of control has terminated. The protocol function associated with the virtual destructor will do the waiting.

ABC++ provides blocking, nonblocking, and future remote procedure calls.

Automatic mutual exclusion. To ensure the integrity of an object, multiple methods of an active object should not be simultaneously invoked. Wegner[24] describes an object with a single thread of control as sequential. Wegner goes on to define quasi-concurrent objects as having multiple threads with only one object active, and concurrent objects having multiple active threads. ABC++ objects are sequential because there is a single thread attached to main, which processes the incoming messages (that is, the method invocations). Only one invocation can be processed at a time, thus guaranteeing mutual exclusion. Users are freed from the burden of managing critical sections themselves through the use of explicit synchronization mechanisms.

Object interaction. An important aspect of a concurrent object-oriented language is the way its objects interact. A natural and easy paradigm of communication is the remote procedure call (RPC) facility.[25] RPC is a mechanism for communication across a network. It is an extension of sequential procedure calls with similar syntax and semantics. A (blocking) RPC blocks the caller until a reply is received. Due to the synchronous nature of RPC, a system providing only a (blocking) RPC cannot fully exploit the inherent parallelism in many applications. A nonblocking (also called asynchronous) RPC sends the message (that is, the method being invoked and its arguments) and returns to the caller object immediately. To maximize parallelism even further, a mechanism is needed that would allow a caller to receive a result from a nonblocking RPC at some future point when the result is needed. Such a mechanism is called a *future*. Futures are commonly used in concurrent object-oriented languages.[6,7,10]

ABC++ provides blocking, nonblocking, and future RPCs. In the present prototype implementation of ABC++, we deal with general-purpose registers in the marshaling of messages, where marshaling refers to the act of packaging the parameters into a message that is then sent to the destination processor. In this case, the number of arguments in methods of active objects is limited to seven (that is, the number of general-purpose registers minus one; one register is used for the this pointer), and their types are limited to simple types and pointers to active objects. We believe this restriction can be eliminated with a more careful implementation of ABC++ and with the use of function templates. Work is presently under way to achieve this goal. The following sections discuss these three kinds of object interaction in ABC++.

RPC interaction. Previously we showed how, with the help of object descriptors, we intercept method invocation and provide location transparency. When a method of an active object is invoked, this call is intercepted and instead a method of the object descriptor (a *protocol* function in ABC++) representing the called object is invoked. In the case of local objects, the protocol function will invoke the "real method" if the object is free and if the called method is currently open. Otherwise, the protocol function will queue up the message for later processing. In the case of remote objects, the call is marshaled to the node where the object resides.

Asynchronous RPC. Many of the concurrent C++-based languages provide asynchronous or future communication (for example, COOL[6]). However, COOL extends the C++ compiler significantly. Similarly, the ES-Kit system[7] introduced futures as the primary method of generating and controlling concurrency. However, ES-Kit managed to provide these facilities by extending the compiler with a nonstandard operator (→). ES-Kit also suffers from the fact that every invocation of every method in an ES-Kit class is gen-

Figure 2 The template for asynchronous method invocation

```
template<class T> class AsynCall {
  private:
    T* actual_pointer;
    virtual FutureResult* af0(...);
    virtual FutureResult* af1(...);
              ⋮
    virtual FutureResult* afn(...);
  public:
    AsynCall(T* p);   // class constructor
    AsynCall( );      // class constructor
    T* operator → ( );
    AsynCall<T>& operator=(T* p);
    AsynCall<T>& operator=(const AsynCall<T>& arg);
}
```

Figure 3 An example of an asynchronous invocation

```
class C: public Task{
  public:
    virtual void f( );
};

main( ){
  C c;      // an instance of C
  AsynCall<C> p = &c; // a smart pointer
  p → f( ); // asynchronous invocation of f
   ⋮
}
```

erated asynchronously. Saleh and Gautron[11] introduced futures into their extended C++ which they call CC++. CC++ requires a preprocessor that renames all user methods so that method invocation can be intercepted. In CC++, all methods that may be invoked with futures must be rewritten (usually in a derived class). This makes the use of futures very inconvenient and awkward.

We employ an approach similar to that in Reference 11 to provide asynchronous method invocation. The template mechanism of C++ is used to define a class of smart pointers, a class of pointers that can be used as regular pointers with the member access operator → to invoke methods of active objects, but with semantics implementing asynchronous method invocation. Figure 2 shows part of the private and public declarations of the AsynCall template.

The AsynCall template provides an overloaded "=" and the appropriate constructors. The nondefault constructor (that is, the constructor that takes one or more arguments) and the "=" operator save the address of the object that is the target of the asynchronous invocation (this address is called actual_pointer in Figure 2). Actual_pointer is an LOD in the case of local objects and an ROD in the case of remote objects. The → operator is also overloaded. It returns a pointer to the AsynCall object itself. This would cause a corresponding virtual method of the AsynCall object (designated as af0, af1, . . . , afn in Figure 2) to be invoked, and not the intended method. The AsynCall virtual method takes care of handling the call on the target object, and returns the address of a placeholder to the caller. The returned value may be discarded by the caller, or made use of in the case of future communication, as next described. In either case, the caller and the called objects proceed concurrently. Figure 3 demonstrates an asynchronous invocation. In this figure, class C inherits from Task, hence its objects are active. In main( ), we declare c as an object of class C and would like to invoke f( ), a method of C, asynchronously. To do this, we instantiate the AsynCall template with class C as its argument. We then declare p as an object of class AsynCall ⟨ C ⟩ and assign it the address of c. Operator "=," as noted above, is overloaded in the AsynCall template, and it saves the address of c. We then invoke f( ) by using the smart pointer p and the member access operator, which is overloaded in AsynCall. The call is immediately returned to the caller, and main( ) continues without blocking.

Future RPC. As shown in Figure 2, the methods of the AsynCall template return a value to the calling program. This return value is designated as FutureResult* in Figure 2. FutureResult is a C++ structure that acts as a placeholder for the actual return value. This return value is discarded in the case of asynchronous method invocations. However, if the calling object expects a return value from an asynchronous invocation, it can simply store the returned value into a future object.

Future objects are instances of the class Future. Once again, we use the template mechanism of C++ to define a class of future objects. This template has parameters based on the type of the expected return value. Figure 4 shows part of the private and public declarations of the Future template. An instance variable of the Future class is

placeHolder of type FutureResult*. This variable is properly initialized by the constructors.

As in the case of AsynCall template, the "=" operator in the Future template is overloaded. This overloaded operator is invoked when the calling object assigns the result of an asynchronous invocation into a future object. The "=" operator returns the future object itself, at which point the calling object proceeds with its activity without having to wait for the actual result. However, that result will usually be needed at some point in the future. The conversion operator is overloaded in the Future template, as shown in Figure 4. This operator is invoked whenever a future object is used in place of the actual return result (for example, the statement y=x in Figure 5). The conversion operator performs more than a simple conversion; it checks to see if the future has been resolved (that is, the asynchronous invocation has returned the expected result). If the future object is unresolved, the conversion operator waits for the completion of the asynchronous invocation.

Once the future object is resolved, the conversion operator returns the expected value, and the calling object proceeds with its activity. The use of future objects is demonstrated in Figure 5. In this figure, class C inherits from Task, hence its objects are active. Method f( ) of C returns an integer value. We like to invoke f( ) asynchronously and collect the return value at a later point when the value is needed. In main( ) we declare a smart pointer as in Figure 3. We then declare a future object x by instantiating class Future ⟨ int ⟩. Method f( ) of c is invoked using p and assigning the return result to x. Operator "=" is overloaded in the Future class, and it returns x. At this point, main( ) continues until the result from f( ) is needed. When the result is needed, the future object x is assigned to y, which is of the same type as the return result of f( ). The conversion operator assigns the returned result to y if the future has been resolved; otherwise, it waits until the future is resolved.

The support for asynchronous and future communication in ABC++ does not require any compiler extensions (as in ES-Kit or COOL) or any preprocessing of the code (as in CC++). Users of ABC++ can choose between synchronous and future method invocations. The future invocation is handled in a simple and natural fashion and without requiring any unreasonable programming conventions (such as rewriting the methods that may be invoked with futures as in CC++).

Figure 4 The future template

```
template⟨class T⟩ class Future {

   FutureResult* placeHolder;
   int resolved;

public:

   Future(const T& val){
      placeHolder=(FutureResult*)val;
   };

   Future( ){ placeHolder = (FutureResult*)0; };

   Future⟨T⟩& operator=(const T& val){
      placeHolder=(FutureResult*)val;
      return *this;
   };
   operator T( );
};
```

Figure 5 An example demonstrating the use of futures

```
class C :public Task{
   public:
      int n;
      virtual int f( );
};

int C::f( ){ . . .; return(n);};
   // f computes for a while, then returns n

main( ){
   int y;
   C c;
   AsynCall⟨C⟩ p = &c;
   Future⟨int⟩ x;

   x = p→ f( ); // an asynchronous call
      which returns some result

   .
   . // do some computation
   .
   y = x; // wait for and then retrieve
      the value returned from p → f( )

}
```

Concluding remarks and future research

In the design and implementation of ABC++, we showed that C++ is powerful enough to support

almost all of the functionality of an extended concurrent language without imposing severe limitations on the use of the language.

One of the prime advantages of ABC++ is the fact that C++ programmers can use a familiar C++ compiler to create active objects and interact with active objects. Users of ABC++ are required to declare as *virtual* all methods of an active object that are invokable by other active objects. We do not believe that this is a major commitment on their part. Although ES-Kit changes the compiler, it still imposes this same programming convention. This requirement can easily be eliminated by a simple preprocessing of the code. However, in this paper our intent was to show how far we can go without changing the language or the compiler, and without employing a preprocessor.

Other limitations in ABC++ are:

- Friend classes and friend functions are restricted to access virtual member functions only.
- No public or static class data members are allowed.
- The types of arguments allowed for methods of active objects as well as return results are restricted to simple types and pointers to active objects. The number of such arguments is limited to seven.
- The number of public member functions is fixed by ABC++ to a relatively large number that can be modified (this number is 32 in the present implementation).

In our current research we are examining the use of function templates to eliminate many of these restrictions. We are also investigating techniques for improving type checking, robustness, and portability of ABC++. Finally, we are examining a shared memory paradigm for inclusion in ABC++. The features of ABC++ follow.

- Support for implicit concurrency through active objects—An active object has its own thread of control and can run simultaneously with other active objects on a shared or distributed memory multiprocessor as well as on a homogeneous cluster of workstations.
- RPC communication—Active objects can communicate through method invocation. An invocation of a method of a remote object is transformed to RPC. Location transparency is provided through the use of object descriptors.
- Selective method acceptance
- Asynchronous communication
- Future communication

All the features of ABC++ discussed in this paper, unless stated otherwise, have been implemented and tested on a cluster of RISC System/6000 processors using TCP/IP as the communication mechanism and on IBM's Scalable POWERparallel System 1 (SP1) using a high-performance message-passing library (EUIH). To our knowledge, ABC++ is the only class library that provides support for all of implicit concurrency, object distribution, selective method acceptance, asynchronous, and future communications.

Acknowledgments

This work has been supported by the Centre for Advanced Studies, IBM Canada Laboratory, and the Natural Sciences and Engineering Research Council of Canada.

Appendix A: A producer and consumer example

The following example illustrates how ABC++ is used. The three classes of Buffer, Producer, and Consumer inherit from Task, hence are concurrent classes. The invokable methods of these concurrent classes are declared as virtual. Instances of these classes are active objects with their own activity. Each class defines a main that becomes the "body" of its active objects. As this example demonstrates, ABC++ syntax is the familiar C++ syntax with no explicit synchronization, mutual exclusion, or message queue management constructs used. The use of the accept statement is demonstrated in the main provided for the Buffer class. For example, when the buffer is empty, an Accept(1, put) is issued. Consequently, all get messages are delayed until a put message is received. After the execution of the first put message, Accept returns and main continues with the statement immediately following Accept(1, put).

```
#include "task.h"
#include "future.h"

class Buffer : public Task {
protected:
  int* array;
  int capacity;
```

```
    int no_in_buffer;
    // number of elements currently in buffer
    void* main( );
  public:
    Buffer(int capacity);
    virtual void put (int item);
    virtual int get ( );
};

class Producer : public Task {
protected:
    int amount
    // number of items to be produced
    Buffer *B;
    void* main( );
public:
    Producer(Buffer* buffer, int no_of_items);
};

class Consumer : public Task {
protected:
    Buffer *B;
    int amount;
    // number of items to be consumed
    void* main( );
public:
    Consumer(Buffer* buffer, int no_of_items);
};

Buffer::Buffer (int size) {
    capacity = size;
    array = new int [capacity];
    no_in_buffer = 0;
}
void* Buffer::main( ) {

    while (1) {
        if (no_in_buffer == 0) Accept(1, put);
        if (no_in_buffer < capacity) Accept(2, put, get);
        if (no_in_buffer == capacity) Accept(1, get);
    }
}

void Buffer::put (int item) {
    array[no_in_buffer++] = item;
}

int Buffer::get ( ) {
    return array[--no_in_buffer];
}

Producer::Producer(Buffer *buffer, int no_of_items) {
    B = buffer;
    amount = no_of_items;
}

void* Producer::main ( ) {
    int item = 0;
    AsynCall(Buffer) ac(B);
    while (item < amount) {
        ac → put(item++);
        // a demonstration of an asynchronous invocation
    }
}

Consumer::Consumer (Buffer* buffer, int no_of_items){
    B = buffer;
    amount = no_of_ items;
}

void* Consumer::main ( ) {
    for (int i = 0; i < amount; i++) {
        int item = B→get( );
        // a demonstration of synchronous invocation
    }
}

    /* in the user's main */

    Buffer *B;
    B = new Buffer(10);
    // a Buffer active object of size 10
    Producer *P;
    Consumer *C;
    C = new Consumer (B,10);
    // a Consumer active object
    P = new Producer (B,10);
    // a Producer active object
```

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of X/Open Co. Ltd. or SUN Microsystems.

Cited references

1. B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, New York (1988).
2. G. R. Andrews and F. Schneider, "Concepts and Notation for Concurrent Programming," *Computing Surveys* **15**, No. 1, 3–43 (March 1983).
3. B. Stroustrup, *The C++ Programming Language* (2nd Edition), Addison-Wesley Publishing Co., Reading, MA (1991).
4. B. Bershad, E. D. Lazowska, and H. M. Levy, "PRESTO: A System for Object-Oriented Parallel Programming," *Software-Practice and Experience* **18**, No. 8, 713–732 (August 1988).
5. P. A. Buhr, G. Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke, "μC^{++}: Concurrency in the Object-Oriented Language C++," *Software-Practice and Experience* **22**, No. 2, 137–172 (1992).
6. R. Chandra, A. Gupta, and J. Hennessy, "COOL: A Lan-

guage for Parallel Programming," *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua, Editors, The MIT Press, Cambridge, MA (1990).
7. A. Chatterjee, A. Khanna, and Y. Hung, "ES-Kit: An Object-Oriented Distributed System," *Concurrency: Software-Practice and Experience* **3**, No. 6, 525–539 (1991).
8. T. W. Doeppner, Jr. and Alan J. Gebele, *C++ on a Parallel Machine*, Report CS-87-26, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912-1910 (November 1987).
9. N. H. Gehani and W. D. Roome, "Concurrent C++: Concurrent Programming with Class(es)," *Software-Practice and Experience* **18**, No. 12, 1157–1177 (1988).
10. D. Kafura and K. H. Lee, "*ACT*$^{++}$: Building a Concurrent C++ with Actors," *JOOP* **3**, No. 1, 25–37 (1990).
11. H. Saleh and P. Gautron, "A Concurrency Control Mechanism for C++ Objects," *Proceedings of ECOOP'91 Workshop on Object-Based Concurrent Programming*, Springer-Verlag, Berlin, pp. 195–210 (1991).
12. H. Saleh and P. Gautron, "A System Library for C++ Distributed Applications on Transputers," *Parallel Computing Action: Selections from Publications of the RXF/LITP Team*, LITP 91-075, RXF Rank Xerox France, Institut Blaise Pascal, 4 place Jussieu, 75252 Paris, Cedex 05, France (December 1991).
13. E. Arjomandi, W. O'Farrell, and I. Kalas, "Concurrency Support for C++: An Overview," *C++ Report* **5**, No. 10, 44–50 (January 1994).
14. *C++ Language System Release 2.0: Product Reference Manual*, Select Code 307-146, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974 (1989).
15. D. Grunwald, *A User's Guide to AWESIME: An Object-Oriented Parallel Programming and Simulation System*, Technical Report CU-CS-552-91, Department of Computer Science, University of Colorado at Boulder, CO 80309-0430 (1991).
16. P. A. Buhr and G. Ditchfield, "Adding Concurrency to a Programming Language," *USENIX C++ Technical Conference*, USENIX Association, Berkeley, CA (1992), pp. 207–224.
17. P. Gautron, "Porting and Extending the C++ Task System with the Support of Lightweight Processes," *USENIX C++ Conference Proceedings*, USENIX Association, Berkeley, CA (1991), pp. 135–146.
18. J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield, "The Amber System: Parallel Programming on a Network of Multiprocessors," *Proceedings of 12th ACM Symposium on Operating System Principles* (1989), pp. 147–158.
19. H. Assenmacher, T. Breitbach, P. Buhler, V. Hunsch, and R. Schwarz, "PANDA: Supporting Distributed Programming in C++," *Proceedings of ECOOP'93—Object-Oriented Programming, Lecture Notes in Computer Science*, Springer-Verlag, Berlin (July 1993), pp. 361–383.
20. M. Shapiro, "Structure and Encapsulation in Distributed Systems: The Proxy Principle," *Proceedings of the 6th International Conference on Distributed Computer Systems*, IEEE Computer Society Press, Paterson, NJ (May 1986), pp. 198–204.
21. A. Dave, M. Sefika, and R. H. Campbell, "Proxies, Application Interfaces and Distributed Systems," *Proceedings of 2nd International Workshop on Object Orientation in Operating Systems*, IEEE Computer Society Press, Paterson, NJ (1992), pp. 212–220.
22. O. Hagsand, H. Herzog, K. Birman, and R. Cooper, "Object-Oriented Reliable Distributed Programming," *Proceedings of 2nd International Workshop on Object Orientation in Operating Systems*, IEEE Computer Society Press, Paterson, NJ (1992), pp. 180–188.
23. C. Nascimento and J. Dollimore, "Behavior Maintenance of Migrating Objects in a Distributed Object-Oriented Programming," *JOOP* **5**, No. 5, 25–33 (1992).
24. P. Wegner, "Dimensions of Object-Based Language Design," *OOPSLA'87*, Object-Oriented Programming Systems, Languages, and Applications, ACM, New York (1987), pp. 168–182.
25. B. J. Nelson, *Remote Procedure Call*, Carnegie-Mellon University, Report CMU-CS-119, Pittsburgh, PA (May '81).

Accepted for publication October 5, 1994.

Eshrat Arjomandi *Department of Computer Science, York University, North York (Toronto), Ontario M3J 1P3, Canada (electronic mail: eshrat@cs.yorku.ca).* Dr. Arjomandi received her M.Sc. and Ph.D. degrees in computer science from the University of Toronto. She is presently Associate Professor of Computer Science at York University, Toronto, Canada. Her research interests are in the areas of object-oriented programming and concurrent object-oriented languages.

William O'Farrell *Centre for Advanced Studies, IBM Canada Ltd., 844 Don Mills Road, North York, Ontario M3C 1V7, Canada (electronic mail: billo@torolab2.vnet.ibm.com).* Dr. O'Farrell is a research staff member at the Centre for Advanced Studies, IBM Canada Laboratory. His research interests are in the areas of parallel computation, object-oriented concurrent systems, and programming languages. He joined IBM in May of 1991, after completing his Ph.D. at Syracuse University. His master's degree is from Queens University at Kingston.

Ivan Kalas *Centre for Advanced Studies, IBM Canada Ltd., 844 Don Mills Road, North York, Ontario M3C 1V7, Canada (electronic mail: kalas@torolab.vnet.ibm.com).* Mr. Kalas is a research staff member at the Centre for Advanced Studies, IBM Canada Laboratory. His research interests are in the areas of programming environments, reactive systems, distributed object-oriented systems, and object-oriented programming languages. He joined IBM in May of 1989.

Gita Koblents *Centre for Advanced Studies, IBM Canada Ltd., 844 Don Mills Road, North York, Ontario M3C 1V7, Canada (electronic mail: koblents@torolab2.vnet.ibm.com).* Ms. Koblents is a developer in the IBM Canada Laboratory. Formerly she was a visiting graduate student at the Centre for Advanced Studies. She is a candidate for an M.Sc. degree in computer science at York University, Toronto, Canada. She is interested in concurrent object-oriented languages. As part of her master's thesis, she contributed to the implementation of ABC++.

Frank Ch. Eigler *Centre for Advanced Studies, IBM Canada Ltd., 844 Don Mills Road, North York, Ontario M3C 1V7 Canada (electronic mail: fche@db.toronto.edu).* Mr. Eigler is

a visiting student at the Centre for Advanced Studies and an undergraduate in computer engineering at the University of Toronto. His interests are in the areas of programming languages, deductive databases, operating systems, and computer networks. He contributed to the implementation of ABC++.

Guang R. Gao *School of Computer Science, McGill University, Montreal, Quebec H3A 2A7, Canada (electronic mail: gao@cs.mcgill.ca).* Dr. Gao received his S.M. and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology in 1982 and 1986, respectively. Currently he is an associate professor at the School of Computer Science, McGill University, Montreal, Canada. He is interested in computer architecture, programming language design and implementation, and parallel systems.

Reprint Order No. G321-5561.