from Queens University at Kingston. He can be reached at the Centre for Advanced Studies, IBM Canada Ltd., 81/894, 895 Don Mills Rd., North York, Ontario M3C 1W3. His email address is billo@torolab2.vnet.ibm.com.

Ivan Kalas is a Research Staff Member at the Centre for Advanced Studies, IBM Canada Laboratory. His research interests are in the area of programming environments, reactive systems, distributed object-oriented systems, and object-oriented programming languages. He joined IBM in May of 1989. He can be reached at the Centre for Advanced Studies, IBM Canada Ltd., 81/894, 895 Don Mills Rd., North York, Ontario M3C 1W3. His email address is kalas@torolab.vnet.ibm.com.

[26] D. Kafura and K.H. Lee, *ACT*$^{++}$: Building a Concurrent C++ with Actors, JOOP, Vol. 3, No. 1, 1990, 25-37.

[27] D. Kafura and K.H. Lee, Inheritance in Actor Based Concurrent Object-Oriented Languages, The Computer Journal, Vol. 32, No. 4, 1989, 297-304.

[28] L. V. Kale and S. Krishnan, *CHARM*$^{++}$: A Portable Concurrent Object Oriented System Based On C++, to appear in OOPSLA'93.

[29] W. Kim and F.H. Lochovsky, Object-Oriented Concepts, Databases, and Applications, ACM Press, 1989.

[30] J. R. Larus, B. Richards, and G. Viswanathan, C^{**}: A Large-Grain, Object-Oriented, Data-Parallel Programming Language, Technical Report UW-1126, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI 53706, 1992.

[31] J.K. Lee and D. Gannon, Object Oriented Parallel Programming Experiments and Results, Supercomputing'91, 1991, 273-282.

[32] M. Lemke and D. Quinian, P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications, CONPAR'92, Proceedings of Second Joint International Conference on Vector and Parallel Processing, 1992.

[33] S. Matsuoka, K. Wakita, and A. Yonezawa, Synchronization Constraints with Inheritance: What Is Not Possible - So What Is?, TR 90-010, Department of Information Science, the University of Tokyo, 1990.

[34] S. Matsuoka and A. Yonezawa, Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, to appear.

[35] M. Papathomas, Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming, Ph.D. Thesis, University of Geneva, 1992.

[36] H. Saleh and P. Gautron, A Concurrency Control Mechanism for C++ Objects, Proceedings of ECOOP'91 Workshop on Object-Based Concurrent Programming, 195-210, 1991.

[37] H. Saleh and P. Gautron, A System Library for C++ Distributed Applications on Transputers, in Parallel Computing Action: Selections from Publications of the RXF/LITP Team, LITP 91-075, RXF, December 1991. Also in Applications of Transputers 91, IOS Press, August 1991.

[38] R. Seliger, Extending C++ to Support Remote Procedure Call, Concurrency, Exception Handling, and Garbage Collection, Proceedings of 1990 USENIX C++ Conference, 1990, 241-264.

[39] B. Stroustrup, The C++ Programming Language (1st Edition), Addison-Wesley, 1986.

[40] B. Stroustrup, A Set of C Classes for Co-routine Style Programming, Technical Report CSRT 90, AT&T, November 1980, revised (1) July 1982, (2) November 1984.

[41] B. Stroustrup, The C++ Programming Language (2nd Edition), Addison-Wesley, 1991.

[42] C. Tomlinson and V. Singh, Inheritance and Synchronization with Enabled-Sets, OOPSLA'89, 103-113.

[43] C. Tomlinson and M. Scheevel, Concurrent Object-Oriented Languages, in [29].

[44] P. Wegner, Dimensions of Object-Based Language Design, OOPSLA'87, 168-182.

About the authors

Eshrat Arjomandi is an Associate Professor of Computer Science at York University. Her research interests are in the area of concurrent object-oriented languages.

Both her Masters and PhD degrees are from the University of Toronto. She can be reached at the Department of Computer Science, York University, North York, Ontario, M3J-1P3. Her email address is eshrat@cs.yorku.ca.

William O'Farrell is a Research Staff Member at the Centre for Advanced Studies, IBM Canada Laboratory. His research interests are in the area 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 Masters degree is

[5] S. Bhatt, M. Chen, C.Y. Lin and P. Liu, Abstractions for Parallel N-body Simulations, DCS/TR-895, Yale University, 1992.

[6] G. Booch and M. Vilot, The Design of the C++ Booch Component, Proceedings of ECOOP/OOPSLA'90, 1990, 1-11.

[7] J.P. Briot and A. Yonezawa, Inheritance and Synchronization in Concurrent OOP, ECOOP'87, 32-40.

[8] P.A. Buhr and G. Ditchfield, Adding Concurrency to a Programming Language, USENIX C++ Technical Conference, 1992.

[9] 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(2), 1992, 137-172.

[10] R. Chandra, A. Gupta and J. Hennessy, COOL: a Language for Parallel Programming, in Languages and Compilers for Parallel Computing, edited by D. Gelernter, A. Nicolau, D. Padua, The MIT Press, 1990.

[11] K. M. Chandy and C. Kesselman, Compositional C++: Compositional Parallel Programming, to appear in the Proceedings of the Fourth Workshop on Parallel Computing and Compilers, Springer Verlag.

[12] J.S. Chase, F.G.Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield, The Amber System: Parallel Programming on a Netwrok of Multiprocessors, Proceedings of 12th ACM Symposium on Operating System Principles, 1989, 147-158.

[13] A. Chatterjee, A. Khanna, and Y. Hung, ES-Kit: An Object-Oriented Distributed System, Concurrency: Practice and Experience, Vol. 3(6), 1991, pp. 525-539.

[14] D. Decouchant, P. Le Dor, M. Riveill, C. Roisin, and X. Rousset de Pina, A Synchronization Mechanism for an Object-Oriented Distributed System, 11th International Conference on Distributed Computing Systems, IEEE 1991, pp. 152-159.

[15] T. G. Dennehy, Class Libraries as an Alternative to Language Extensions for Distributed Programming, SEDMS III, Symposium on Experiences with Distributed and Multiprocessor Systems, USENIX Association, 1992, 313-326.

[16] T.W. Doeppner Jr., and Alan J. Gebele, C++ on a parallel machine, Report CS-87-26, Department of Computer Science, Brown University, November 1987.

[17] Encore Parallel Threads Manual, Encore Computer Corporation 724-06210, May 1988.

[18] S. Frolund, Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages, ECOOP'92, 1992, 185-196.

[19] P. Gautron, Porting and Extending the C++ Task System with the Support of Lightweight Processes, USENIX C++ Conference Proceedings, 1991, 135-146.

[20] N.H. Gehani and W.D. Roome, Concurrent C++: Concurrent Programming with Class(es), Software-Practice and Experience, Vol. 18(12), 1988, 1157-1177.

[21] A. S. Grimshaw and J.W.S. Liu, Mentat: An Object-Oriented Data-Flow System, Proceeding of the 1987 Object-Oriented Programming Systems, Languages and Applications, OOPSLA'87, 35-47.

[22] A.S. Grimshaw, Easy-to-use Object-Oriented Parallel Processing with Mentat, Technical Report No. CS-92-32, Department of Computer Science, University of Virginia.

[23] 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.

[24] T. L. Hansen, The C++ Answer Book, Addison Wesley, MA, 1990.

[25] J.M. Jezequel, F. Bergheul and F. Andre, Programming Massively Parallel Architectures with Sequential Object-Oriented Languages, PARLE'92, Parellel Architecture and Languages Europe, 1992, 329-344.

archy, which uses inheritance to reuse implementation, is quite restrictive with respect to derivation. The basic unit of activity, implemented by the class *task*, allows only one level of subclassing. Task objects communicate by explicitly manipulating message queues.

LWP/C++ (our nomenclature) [19] is an implementation of the AT&T's task library on top of the Sun LightWeight Process library. As such, LWP/C++ represents a departure from the original co-routine task paradigm of AT&T into the user-level (uniprocessor) preemptable threads. LWP/C++ also extends the AT&T FIFO scheduling model to include LIFO and user-controlled scheduling.

EPT/C++ (our nomenclature) [16] is a tasking package modeled after the AT&T library, but built on top of a threads library known as Threads, which was the basis for the Encore Parallel Threads library (EPT) [17]. EPT/C++ is one of the earliest C++ class libraries supporting true concurrency. In addition to supporting a Task class, which if used as a base class causes a thread to be created for the constructor of the derived class, EPT/C++ also supports a Monitor class that can be used to provide guarding for methods of derived classes. However, entry to and exit from monitors are explicitly managed by the users through *enter* and *exit* routines. Threads are explicitly managed by operations such as *wait* and *wakeup*. As in AT&T's task library, EPT/C++ allows only one level of subclassing. Communication is primarily through explicit message queues.

PRESTO [4] is a class library for the creation and management of threads. There are PRESTO classes for each concurrency concept, principally threads, locks, and monitors. Creating a thread object creates, within the thread-object data area, the structures necessary for a thread's existence (for example, a stack), although, notably, it does not start a new thread running. The thread *start* method must be called for this to happen. Similarly, creating a lock object allocates storage for the lock, which is then manipulated via lock methods. Both spinning locks and relinquishing locks are available. Monitors are created and manipulated in the same fashion. In PRESTO, monitor routines are not physically associated with the monitor control object. The programmer must explicitly indicate when a monitor routine is being entered (via a call to an entry method) and when it is being exited (via the *exit* method).

The assumed architecture for PRESTO is shared memory. PRESTO has no specific constructs for object interaction, even in the guise of C++ method invocation; communication happens solely through the use of shared variables. PRESTO comprises a set of low-level routines for shared-memory concurrency. Its principal advantage is high performance on such architectures.

AWESIME [23] is a thread library, not unlike PRESTO, with additional classes to facilitate process-oriented simulation. It is designed to simplify portability to other architectures. Users can create their own customized threads by inheriting from an abstract class called *THREAD*. The "body" of a thread is specified by a virtual *main* member function. Unlike the AT&T task library, it allows arbitrary levels of subclassing from *THREAD*. However, a thread does not start execution until it is explicitly triggered.

5 Summary

In this paper, we reviewed and analyzed some of the C++-based concurrent systems. Issues characterizing concurrent object-oriented systems and techniques used in adding concurrency to C++ were presented. We compared concurrent extensions to C++, and concurrent libraries for C++ where concurrency is kept outside of the language.

References

[1] G. Agha, ACTORS: A Model of Concurrent Computation in Distributed Systems, MIT Press, 1986.

[2] E. Arjomandi and W. O'Farrell, Active Objects in C++: How Far Can We Stretch the Library Approach? IBM Canada Laboratory Technical Report, TR-74.116, 1993.

[3] AT&T C++ Language System Release 2.0: Product Reference Manual, 1989, Select Code 307-146.

[4] B. Bershad, E. D. Lazowska and H. M. Levy, PRESTO: A System for Object-Oriented Parallel Programming, Software-Practice and Experience, Vol. 18(8), 713-732, August 1988.

An object may have several threads in it at one time, but if the object is a monitor or a task, only one can be active at a time. Hence task and monitor objects are quasi-concurrent.

μC++ supports inheritance in a limited way. Classes defined with extended types (co-routines, monitors and tasks) can inherit only from classes of the same type. Among tasks, when multiple bodies are to be inherited, the one selected to be the active body is the one declared the deepest in the class hierarchy.

4 Overview of Concurrent Library Support Systems for C^{++}

We identify two categories of concurrent library support systems for C++: *implicit* and *explicit*. In the following, we discuss some of the C++-based concurrent libraries in each of these categories. Users of concurrent libraries must observe certain protocols with varying degree of severity. We discuss the limitations of each of the libraries. Because of the lack of compiler support, concurrent libraries do not provide any facilities to minimize the effect of the conflict between inheritance and synchronization code.

4.1 Implicit

Concurrent libraries in this category use the object-oriented facilities to encapsulate concurrency at the object level.

As outlined in [8,9], encapsulating concurrency at the object level and providing the concurrent library with sufficient flexibility and power poses many challenging problems that are difficult to solve without compiler support. These problems include:

- Creating active objects in the presence of arbitrary levels of inheritance and without requiring the users to trigger thread activation.
- Preventing invocation of methods in newly created objects before construction is completed.
- Preventing object destruction until the object's thread of control has terminated.
- Preventing recursion deadlocks.
- Selective method acceptance.
- Object distribution and migration.
- Object interaction by method invocation in a natural and transparent fashion.

Many of the concurrent class libraries for C++ have not solved all of the above problems and instead have imposed a variety of restrictions on users [3,16]. These restrictions can include limiting the number of inheritance levels to one, explicit thread management through the use of a *start* routine; requiring explicit message queues in object interaction; and explicit management of synchronization and mutual exclusion.

ABC++ [2] is an example of a class library that has solved most of the above problems while minimizing the restrictions on users.

ABC++ [2] (Active Base Class) supports active objects that possess their own thread of control and can run simultaneously with other active objects on a shared or distributed memory multiprocessor. In the case of distributed memory, an invocation of a method of a remote object is automatically transformed into a remote procedure call (*rpc*). Location transparency is provided through a novel use of *object descriptors* (also called proxies). Object distribution and selective method acceptance are also supported. ABC++ provides much of the functionality offered by a new or extended language without imposing a significant burden on the users. Users of ABC++ must observe the requirement that all methods of an object that are invokable by other objects must be declared as *virtual*.

4.2 Explicit

The class libraries in this category use the *abstract data type* approach to provide a set of concurrency primitives with which the users can write concurrent programs. Some of these libraries support active objects, however, the control of synchronization and mutual exclusion is more visible at the user interface level. Also note that most of these libraries focus on the shared memory style of concurrent programming.

AT&T's [3] task library is perhaps the earliest attempt at providing some level of concurrency support (through co-routines) in a C++ class library. It originated in the early days of C++ (circa 1984) [40] and is discussed in [39] and [24]. Its class hier-

Communication in COOL is through shared memory and discussions of object interaction or selective message acceptance do not apply. The language in its present implementation has not integrated some features of C++ (for example, *friends*, *virtual* functions, overloading, inheritance) with the new concurrency constructs.

3.2 Explicit Concurrency

In these languages, there is less integration of concurrent and object-oriented paradigms. The creation and control of concurrency is more visible at the user interface level. In these languages, programmers are concerned with creation and manipulation of two units of execution: objects and concurrency units. Concurrent C++ [20] and Extended C++ [38] are examples of such languages.

Concurrent C++ [20] is developed at AT&T Bell Laboratories. It is an integration of AT&T's Concurrent C with C++. Concurrency is achieved by creating many communicating processes. Concurrent C++ uses the term *transaction* (a form of rpc) for process interaction. Transactions can be synchronous (blocking) or asynchronous (non-blocking). Asynchronous transactions are one-way communications and return no results to the caller. Transactions can also be used for process synchronization. Processes can selectively accept messages by using an Ada-like *select* statement. The selection can be based on the internal state of the process or on the arguments of the transaction. Facilities such as priority specification and waiting for multiple events are also provided. Classes of C++ can be used to hide implementation details and enforce the visible external protocol of a process.

Extended C++ [38] is a superset of C++ with support for concurrency, exception handling (non-ANSI compiant) and garbage collection. It was developed to facilitate distributed OOP in the critical care environment for a hospital. Concurrency can be created by defining many light-weight processes (threads) executing in the same address space (interleaved concurrency) or by invoking member functions of remote objects. A member function declared as *threadable* can be used to begin a new process. A *threadit* operator defines the point in the program where the thread is spawned. The operand for this operator evaluates to a call to a threadable member function. Mutual exclusion is achieved by the use of *regions.* A region is executed by one thread at a time. *Release* and *trigger* statements can be used to suspend or awaken a thread. The built-in class type *condition* allows users to define conditional variables that would make a thread wait on a condition queue until a specific user-defined condition is satisfied.

Communication among remote objects is by *rpc.* Only member functions that are declared *remotable* can be invoked remotely. A class with a *remotable* member function is called a *remotable* class. Extended C++ does not provide any explicit message acceptance facility; however, condition variables can be used to suspend the execution of a method until a particular condition is met.

3.3 Mixed Concurrency

μ**C++** [9] provides concurrency by creating active objects. μC++ extends C++ by introducing new type constructs for co-routines, monitors, co-routine monitors, and tasks. Task objects are active and are the only μC++ objects with their own thread of control. The authors argue that while it is possible to simplify the language by just providing task objects, creating co-routine and monitor objects out of class and task objects is difficult and inefficient. We have included μC++ in the mixed category, since while one can choose to use only task objects that encapsulate concurrency, the language provides other constructs that allows the programmer to decide among various concurrency constructs.

Active objects have a distinguished method called *main* which is the active body of the object. The assumed memory model is shared memory. Object interaction is by method invocation, and may be thought of as blocking, in the sense that a calling thread may be blocked by virtue of mutual exclusion. However, the method arguments are stored only in the caller's stack frame and are not copied (the shared memory model makes this possible). Synchronization control in μC++ is achieved by an Ada-like *accept* statement.

When a task method is invoked, it is the caller's thread that enters the method. Unlike Concurrent C++ transactions, the processing of the call can be deferred if required, by waiting on a condition queue. When a thread places itself on a queue, it releases exclusive access to the object, allowing another thread to proceed. To a certain extent, this capability makes up for the inability of the accept statement to examine actual arguments.

sign of CC++.

CHARM++ [28] is a portable extension of C++ that runs on many MIMD machines. Active objects in CHARM++ are called *chares*. Chares have exactly one thread of control. A CHARM++ program consists of a few modules, which can be separately compiled. Each module may include message declarations, *chare* class definitions, *branched chare* class definitions (described below), shared object definitions, global functions and hierarchies of C++ code (with restricted use of global and static variable). A branched chare is a replicated chare that has a branch on all processors. Each branched chare has its own local data. Branched chares facilitate data-parallel operations. Each chare object is identified by a unique *handle*. The handle of a branched chare object is the same for all its branches.

Communication among chares is by asynchronous message passing. A *message* declaration in a CHARM++ module defines the data that constitutes a particular message. Chares and branched chares define the types of services that they provide to other chares through *entry* point declarations (i.e. entries define the protocol of the active object). Entry points are like functions with no return values and exactly one argument which is a pointer to a *message*. CHARM++ prohibits the use of public member functions in chares, thereby enforcing their protocol and communication by message passing. Branched chares can have public data and member functions. Access to these public members is handled by the local branch of the branched chare.

CHARM++ supports dynamic binding, overloading, dynamic load balancing, and a variety of message scheduling schemes. CHARM++ allows multiple inheritance among concurrent classes. CHARM++ does not provide any explicit facility for selective message acceptance.

Mentat [21,22] supports medium grain parallelism by providing a portable set of run-time facilities and language abstractions to encapsulate concurrency at the object level. A Mentat program consists of C++ and MENTAT classes. Instances of MENTAT classes run in separate address spaces and have their own thread of control. Each Mentat object is given a unique, system wide identification. Mentat objects are treated exactly like C++ objects. Invocation of a method of a Mentat object is automatically converted to a (*future* like) non-blocking rpc. Mentat supports an ada-like *select/accept* statement. Selection can be based on the state of the object as well as on the arguments of the caller. A Mentat class may be declared as PERSISTENT or as REGULAR. Instances of PERSISTENT classes maintain their state between invocations.

The underlying computational model of Mentat is a *macro data flow model*. An executing program is a directed graph in which the nodes are the Mentat object member function invocations and the edges are the data dependencies found between the two invocations. The compiler generates the code to construct and execute the data flow graph.

2. Languages with Concurrency Encapsulated at Class Level. In this class of languages, objects are not active; however, there is still a strong integration of object-oriented and concurrency paradigms. Concurrency, synchronization and mutual exclusion are largely implicit and encapsulated within the class definition. Concurrency is generally introduced by allowing asynchronous method invocations on passive objects. Thread creation is at the time of method invocation. COOL [10] is an example of such a language.

COOL [10] is an extension of C++ with additional constructs to create concurrency. COOL supports medium to large-grain parallelism. Concurrency is created either by executing methods on different objects or by executing many methods on the same object. *Private* or *public* member functions can be declared as *parallel*. Method invocation of parallel methods is asynchronous, and future variables are used to receive the results of the calls. If a public member function requires exclusive access to an object, it is declared as *mutex*. Introducing mutual exclusion at the function level is more flexible than introducing them at the object level, as in monitor objects. In monitors, all methods are mutex. The use of a mutex function may result in a deadlock. A mutex function executing on an object may need the result of another function executing on the same object. If the first function blocks on this object, a deadlock occurs. To avoid this, COOL has a *release* statement that can be used by a mutex function. The use of *release* blocks the mutex function and releases the object so that other functions can access it. Besides mutex functions and future variables as synchronization mechanisms, COOL provides *locks* for synchronization at a finer granularity.

which have addressed the conflict between inheritance and synchronization code and have provided partial solutions to minimize its effect, none of the other languages have addressed this issue (although many of them allow inheritance among concurrent classes without any special attention to this conflict).

3.1 Implicit Concurrency

We further subdivide this category into two classes: (1) concurrency created and controlled by active objects, (2) concurrency encapsulated at the class level.

1. Languages with Active Objects. These languages naturally lend themselves to a distributed memory model of computation. An active object may support intra-object concurrency either in a quasi-concurrent fashion, as in a monitor, or by supporting many concurrent threads. ACT++ [26], CHARM++ [28], Mentat [21,22], and Saleh and Gautron's CC++ [36,37] are examples of a C++-based concurrent language in this category.

ACT++ [26] is a shared-memory, actor-based language. For more detail on actor models, the reader is referred to [1,43]. Instruction-level concurrency, often found in actor languages, is not supported in ACT++. The language supports medium to large-grain parallelism. This is achieved by using primitive data and control structures of C++ as well as C++ objects (passive objects) in the construction of an active object. The identity of the passive objects should be known only to the containing active object, although this is enforced only by convention.

ACT++ programmers use three classes – ACTOR, Mbox, and Cbox – to create actors (active objects), mail boxes for actors, and future variables for receiving replies, respectively. Concurrency is created either by creating new actors with their own thread of control (inter-object concurrency) or by defining new behaviors for existing actors through the use of the *become* statement (intra-object concurrency). In an actor-based language, the *become* facility is both a synchronization mechanism as well as a mechanism for changing the state of an actor (actor operations are side-effect free). Each behavior of an actor has its own thread of control and proceeds in parallel with the original actor. An actor, after receiving a message, defines a replacement behavior that may have a different message protocol and new values for its instance variables (called acquaintances in the actor model). An actor prevents the processing of other messages until the replacement behavior is ready. The replacement behavior proceeds with the processing of the next message in the mail queue. An active object in ACT++ may be considered as a set of states, with each state responding to a particular set of messages. Therefore, selective method acceptance is supported by the ability of an actor changing its protocol through defining a replacement behavior. However, the message acceptance is not parametrized with the arguments of the invoked method.

Kafura and Lee [27] show that ACT++, without additional facilities, does not support inheritance between active objects. They then introduce the notion of *behavior abstraction*, to aid the inheritance process. This abstraction facility provides only a partial solution [33] in the context of actor-based languages that use interface-control for synchronization. Sequential class reusability in ACT++ is difficult because of its concurrency control mechanisms.

Saleh and Gautron's CC++ [36,37] introduces concurrency in C++ partly by language extension and partly by class library support. To facilitate inheritance of synchronization code and sequential class reuse, CC++ extends C++ by adding a concurrency control mechanism based on a concept called *conditional wait*. This is implemented by extending the set of access specifiers in C++ (public, private, and protected) to include a *delay* access specifier. Methods with synchronization conditions are declared within the scope of a delay specifier. The delay keyword is also used to declare conditions under which the execution of a method may be delayed. The synchronization rules may be parametrized with the arguments of the invoked method. The separation of synchronization rules from the method definition is a step toward minimizing the effect of the conflict between inheritance and synchronization code. Intra-object concurrency is supported by declaring some methods as constant as well as by using the delay declarations to describe under what conditions methods of an object may be invoked in parallel.

Support for *rpc*, *future rpc*, and location transparency is provided by a class library. We have included CC++ with the concurrent systems that extend C++ because some of the concurrent support is provided by language extension. The CC++ compiler generates extra code when the delay declarations are parsed. The parsing also converts passive objects to active. Code reuse is one of the premises in the de-

- **Sequential** objects have exactly one thread of control.

- **Quasi-concurrent** objects act like a monitor. They possess many threads of control, but only one thread is active at a time. These objects allow a thread to be suspended while waiting for a condition to be met.

- **Concurrent** objects allow many threads to be active simultaneously. Thread creation can take place in one of two ways: (a) by method invocation or message reception; (b) by executing a construct, triggered by the object.

Inheritance. As mentioned earlier, concurrency and object-oriented features are not orthogonal. A well known problem in COOPLs is the conflict between inheritance and synchronization mechanisms. Synchronization mechanisms maintain the integrity of objects by specifying under what conditions methods of an object may be invoked. One of the difficulties in writing concurrent programs is the specification of synchronization code. It is therefore most desirable to avoid re-writing of such code and to exploit inheritance for code reuse.

The conflict between inheritance and synchronization mechanisms has been reported by many researchers [7,14,18,27,33,34,35,42,44]. Tomlinson and Singh [42] define a synchronization mechanism as interfering with inheritance if local changes in the inheritance hierarchy (for example, adding a new method to a subclass) require changes in the synchronization constraints elsewhere in the hierarchy. Because of this interference, many concurrent languages do not support inheritance or support only a limited inheritance. Concurrent object-oriented languages use one of two ways to control concurrency: *centralized* or *decentralized* [27]. In the centralized approach, one procedure controls the concurrency by indicating when it is ready to accept a particular message. In such languages, the addition of a new method always requires changes in the procedure that controls concurrency. Furthermore, Matsuoka, Yonezawa, and Wakita [33,34] show that when there are complex interactions between the newly added method and the existing methods, the addition of a new method may require changes in the previously written methods. In the decentralized approach the concurrency control is distributed among methods. This can be achieved either by the use of entry and exit protocols (as in Extended-C++ [38]) or by objects dynamically changing their interface (as in ACT++ [26]). In the case of entry and exit protocols, subclasses must observe such protocols imposed by the superclasses, hence violating the principles of encapsulation. In the case of objects changing their interface, addition of a new method requires changes in the superclass methods.

Many proposals have posited solutions to this problem, with only a partial success [14,18,27,34,35,42]. Matsuoka, Yonezawa, and Wakita [33,34] formally prove the conflict between inheritance and synchronization mechanisms and present counter examples to many of the proposed solutions. Present COOPLs either do not address this conflict or provide a partial solution to minimize its effect.

Sequential Class Reusability. The success of a C++ system in a development environment to a large degree depends on class reusability. A concurrent C++ language or library should provide provisions for using "properly" designed hierarchies of C++ classes. By "properly" we mean that the philosophy of object-oriented design is observed (for example, objects are fully encapsulated), and special attention is paid to the possibility of class reuse in a concurrent setting. One such example is Booch's class library [6]. However, even in the case of properly designed C++ hierarchies, the designer of the concurrent C++ system has to provide for class reuse without requiring any editing of previously written C++ classes. Many of the C++-based concurrent systems do not support sequential class reuse.

3 C^{++}-Based Concurrent Languages

We identify three categories of concurrent object-oriented languages. In the first category, concurrency creation, synchronization and mutual exclusion mechanisms are *implicit* and encapsulated within the object or the class definition. The second category is the *explicit* category, in which creation and control of concurrency is more visible at the user interface level. The third category is the *mixed* category, where both implicit and explicit concurrency mechanisms are provided. In the following, we discuss some of the C++-based languages that lie within each of these categories. With the exception of Saleh and Gautron's CC++ and ACT++, concurrent C++-based systems are covered. Notable examples are [12,13,11]. Similarly, C++-based data-parallel languages, such as PC++ [31] and C^{**} [30], and the application specific concurrent systems such as [5,25,32] are not covered.

Section 2 discusses issues of concern in concurrent object-oriented languages. Section 3 reviews concurrent extensions to C++. Concurrent libraries for C++ are reviewed in Section 4. A brief summary is provided in Section 5.

2 Issues in Concurrent Object-Oriented Programming Languages

The following issues characterize a concurrent object-oriented programming language (COOPL).

Memory Model. Most concurrent languages can be assigned to one of two camps: distributed memory or shared memory. In a distributed memory model, processors have exclusive access to their own local memories and communicate by passing messages. In a shared memory model, all processors share a large global memory and communicate via synchronized access to shared variables. Object-oriented programming is often regarded as a form of programming with messages, even on sequential machines. Specifically, invoking a particular method of an object can be seen as sending a message to the object. Communicating with an object by sending it a message facilitates the enforcement of encapsulation.

Because of the close analogy of OOP and message passing, COOPLs would seem to fit naturally into the distributed memory model. However, because of weak encapsulation in C++, concurrent C++ languages on distributed memory models face the problem of pointers to data members being passed to objects in different memory spaces.

Object Interaction. An important characteristic of a concurrent object-oriented language is the way objects/processes interact. Two paradigms of object interaction can be identified [43]. In the first paradigm, objects interact with messages using *send* and *receive* primitives. An interaction style consisting of a non-blocking send primitive and a blocking receive primitive is called *asynchronous* message passing. *Synchronous* message passing uses blocking send and blocking receive.

The second paradigm of interaction consists of two primitives: *call* and *reply*. This paradigm is an extension of sequential procedure call and is commonly referred to as remote procedure call (*rpc*). In a *rpc* communication, objects request services of remote objects in a natural and transparent fashion. We identify two styles of *rpc*s. A (blocking) *rpc* refers to a style of interaction in which the caller object blocks as soon as the call is issued until a reply is received.

To maximize concurrency, an object may want to request the services of another object and proceed with its own activity until the result of the request is needed. In the rpc paradigm of interaction, this is achieved by the call immediately returning what is called a *future* object, thus allowing the caller to proceed. The caller object proceeds with its own activity until it needs the result of the call, at which point it either blocks, if the *future* object has not received the reply from the called object, or receives the reply and proceeds with its activity. This style of interaction is called *future rpc*.

Active/Passive Objects. An object encapsulating data structures, operations, communication, and synchronization mechanisms is called an *active* object. Namely, the notions of *object* and *process* are unified into a single notion of an *active* object (object = process). An *active* object possesses its own thread(s) of control. In contrast, a passive object does not have its own thread of control and has to rely on active objects containing it or other synchronization mechanisms, such as locks, monitors, etc, to ensure its integrity.

Selective Method Acceptance. A COOPL receives much of its flexibility and power from the ability of objects to selectively accept which requests to serve and which to delay for later processing. Accept statements vary in form and power. An object may accept a particular message based on its internal state, parameters provided in the message, or the sequences in which messages may be accepted.

Intra Object Concurrency. If a concurrent program is to model real-world problems, a question arises with respect to the level of concurrency within an active object. This has been the subject of much discussion. Wegner [44] classifies COOPLs into three categories, depending on the level of internal object concurrency:

Concurrency Support for C++: An Overview

Eshrat Arjomandi, York University
William O'Farrell and Ivan Kalas, IBM Canada

Abstract

Many attempts have been made in adding concurrency to C++ either by extending the language or through the use of a class library. This paper reviews and analyzes some of the concurrent C++-based systems. We study the various approaches taken by these systems in adding concurrency to C++ and how these approaches interact with the object-oriented paradigm.

1 Introduction

Concurrency has been the subject of much research for over three decades. Traditionally, concurrency is concerned with threads of control and problems of synchronization and mutual exclusion. Object-oriented programming (OOP) languages provide a natural facility for modeling and decomposing a problem into *self-contained* entities called objects. An object is self-contained in the sense that it has exclusive control over its own internal state and communicates with the outside world through message passing. Thus it appears that the integration of the two paradigms of OOP and concurrent programming is promising. However, concurrency and object-oriented issues, as they may appear at first glance, are not orthogonal, and the integration of the two poses many challenging problems.

Two approaches can be used in the integration of the two paradigms of concurrency and OOP. In the first approach, language extensions or new concurrent languages are introduced, with the concurrency constructs being part of the language. The second approach uses OOP in developing a library of reusable abstractions that encapsulate the lower level details of concurrency (for example, architecture, data partitions, communication and synchronization). These libraries keep the concurrency constructs outside of the language. Programmers can use these libraries through well defined interfaces, using the existing language constructs.

There is much debate among language designers about the approach to take in the integration of concurrency and OOP [8,15]. The library approach keeps the language small, allows the programmer to work with familiar compilers and tools, provides the option of supporting many concurrent models through a variety of libraries, and eases porting of code to other architectures (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. On the other hand, new or extended languages can use the compiler to provide higher level constructs, compile-time type checking, and enhanced performance.

C++ is increasingly becoming the language of choice among developers. It is therefore not surprising to see the many attempts that have been made to add concurrency to C++ [2,3,4,9,10,16,19, 20,21,23,26,28,36,38] . This paper reviews and analyzes some of the concurrent C++-based systems. We study the various approaches taken in adding concurrency to C++ and how these approaches interact with the object-oriented paradigm. We give special attention to concurrent C++ systems that have used the assistance of a task library to add concurrency to C++ and review some of the existing concurrent class libraries for C++, outlining their features and limitations. Due to space limitations, not all