http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 5 (3): 601-611, 2006
ISSN 1812-5638

A Survey of Concurrent Object-oriented Languages (Cools)

[1]Kalim Qureshi and [2]Paul Manuel
[1]Department of Mathematics and Computer Science,
[2]Department of Information Sciences, Kuwait University, Kuwait

Abstract: The progress in hardware and networking has changed the computing environment from sequential to parallel. During the last decade object oriented programing has made a widespread influence. Many attempts have been made to combine both developments. The main objective was to provide the advantages of object oriented software design at the increased power of parallel machines. This survey covers the well known characteristics of both object oriented paradigm and parallel programming and then mark the design space with possible combinations by identifying various interdependencies and key concepts. The survey presents the characteristics and feature tables for 111 proposed languages.

Key words: Concurrent object-oriented programming languages, parallel and object-riented paradigms, parallel paradigm, survey

INTRODUCTION

Concurrent object-oriented programming languages (COOL) focus on the abstraction and encapsulation power of abstract data types on managing the complexities of concurrency and distribution. In particular, pure fine-grained concurrent object-oriented languages (as opposed to hybrid or data parallel) provides the programmer with a simple, uniform and flexible model while exposing maximum concurrency. While such languages promise to greatly reduce the complexity of large-scale concurrent programming, the popularity of these languages has been hampered by efficiency which is often many orders of magnitude less than that of comparable sequential code.

The increasing use of parallel machines has exacerbated the longstanding tension between high-level and low-level programming languages. Though high-level languages ease the task of expressing a computation, advocates of low-level languages argue that detailed control is required to achieve efficiency. Arguably, moving to parallel systems increases both the complexity of programming and the importance of achieving high efficiency. Thus, determining what high level features can be supported efficiently and how to implement them efficiently is an important topic of research.

The larger problem of achieving good parallel performance requires both generation of efficient sequential code and data locality. We explore the elimination of object-orientation and concurrency control costs in the generated code. Even the best implementations incur tens to hundreds of instructions for each method invocation (America, 1987) due to the cost of managing a distributed memory. Furthermore, the high procedure call frequency typical of object-oriented programs not only magnifies the method invocation overhead, it also reduces the benefits of traditional optimization.

Basics of object-orientation: An object is the basic programming entity. It takes up space in memory and has an associated address. The object stores a 'state' and offers a set of methods for meaningful operations on that state. A language with objects provides data abstraction (data encapsulation) if the state encoding can be hidden so that it can only be accessed through methods instead of direct access to instance variables.

A class is an implementation of a set of objects. Objects of the same class share the same implementation. A class determines a concrete type, i.e., the interface of methods that are offered by that implementation.

Languages with objects, but without the notion of classes, are called object-based languages (Agha, 1986). Languages that offer both objects and classes are referred to as class-based languages. Object-based languages with a mechanism to clone objects, i.e., to make several objects adhering to a common interface and implementation, are called prototype-based languages (Turcotte, 1993). In class-based programming languages, classes hide information regarding their internal details behind a well defined interface and hence support a modular system design. Inheritance is the essential feature that turns class-based languages into object-oriented languages (Meyer, 1988).

Corresponding Author: Dr. Kalim Qureshi, Department of Mathematics and Computer Science, Kuwait University, Kuwait

An object-oriented language is said to offer multiple inheritance if a new class can inherit from the implementations of more than one (unrelated) ancestor or if a new class can be in the type hierarchy below two different types. For full extensibility, however, the two additional characteristics (Bal *et al*., 1989). Polymorphism and dynamic binding have to be offered by the language.

COOL PROGRAMMING PROBLEMS

The common problems when concurrency is added to object oriented language. Most of these problems can be avoided by appropriate programming style or language design.

Parallel Performance

1. Fan-out: In general it is inefficient to spawn activities sequentially since on a machine with p processors it takes p steps until all processors are busy. In COOLs that only support the creation of a single new activity the programmer can always use a binary creation tree to reduce the time to log(p) steps until all p processors are busy. However, such code is often hard to read and programmers tend to avoid it. For increased expressiveness, COOLs should therefore offer spawning constructs with high fan-out.
2. Intra-object concurrency: In many COOLs, objects are implemented as monitors without intra-object concurrency so that only one method is executed at a time while concurrent invocations are delayed. Since in general such delays are inefficient, COOLs should allow intra-object concurrency, i.e., it should be possible to invoke several methods of an object concurrently.
3. Locality: On distributed memory parallel machines, good parallel performance can only be achieved if objects and activities are co-located in the same node to avoid slow remote access.

Inheritance anomalies: The concept of inheritance is meant to refine certain aspects of a class while keeping other aspects stable and reusing them. However, in an implementation with callee-side coordination, a class implementation contains instance variables, code that implements the intended functionality and code that implements the coordination constraints. In general, there is a high interdependence between coordination constraints of different methods and instance variables (Bal, 1991). Concurrency coordination and functionality is intimately interwoven. Because of this interdependence, methods often cannot be refined in subclasses without affecting other methods due to modified coordination constraints.

Expressive coordination constraints: Callee-side coordination needs mechanisms to express so-called proceed-criteria that specify whether a method invocation can proceed or must be delayed. As discussed in the previous section, proceed criteria need to be isolated and separable to reduce inheritance anomalies.

INITIATING CONCURRENCY

The initial question of parallel programming is how to initiate parallel execution. In this section we present various proposals for expressing parallel execution in object-oriented programming languages and discuss whether these mechanisms are appropriate (Yaoqing and Kwong, 1993; Wyatt *et al*., 1992).

Automatic parallelization: The sequential representation provided by the programmer is automatically converted into a parallel form. Conceptually, automatic parallelization fits well to object-oriented programming languages since it does not visibly interfere with existing language characteristics.

Fork, join and equivalents: This section covers constructs that start exactly one new concurrent activity. This activity is not bound to objects but can operate on the data structures in the same way as the activity that executed the construct. Table 1 gives the overview of Fork-Join and Equivalent Category of COOLs.

Fork and join: A method can be invoked with the fork statement, but after the start both the invoking and the invoked method proceed concurrently. Together with fork, often a join statement is introduced. The process that executes the join blocks unless/until the forked method has terminated. There can be several join statements that refer to a single fork but only one may be used at run-time. Basic fork and join restrict parallel performance due to limited fan-out and are affected by the library problem if fork is provided by a separate library. Callee-side coordination is needed as a programming style to achieve encapsulation. Fork and join do not obey the single-entry single-exit paradigm. Unless used with discipline, the programs are speckled with fork and join statements.

Asynchronous call and future: Several COOLs provide an asynchronous method call that is equivalent to the fork statement as long as there are no return parameters. The programmer cannot determine when the asynchronously called method has terminated since there is no join.

Table 1: COOLs in fork-join and equivalent category

COOL	Fork	Async. call	1st class future	Post-processing	Misc. features
ABCL/x		✓	✓	✓	
Acore		✓		✓	
ACT++		✓	✓	✓	First, last, queue
Act1		✓	✓	✓	Once, queue
Actalk		✓		✓	
Actorpsace		✓		✓	
Actra				✓	
A-NETL		✓	✓	✓	
Amber					Thread object
ASK		✓		✓	
A'UM		✓			
Cantor		✓		✓	
CEiffel		✓			Wait by necessity
CHARM++		✓			
CLIX		✓		✓	
Compos. C++					Spawn command
Conc. aggregates		✓		✓	
Conc. smaltalk		✓	✓	✓	
cooC		✓			Wait by necessity
COOL/Chorus	✓				
COOL/NTT		✓			
COOL/Stanford		✓			
Coral		✓			
Correlate		✓			
CST		✓	✓		
Demeter					Thread object
Distr. C++					Thread object
Distr. Eiffel		✓	✓		
Distr. smalltalk-object	✓				
Distr. smlltalk-process	✓				
DOWL					Thread object
DROL		✓		✓	
Ellie		✓	✓		
ES-kit		✓			Manual future
ESP		✓	✓		
FOG/C++		✓			Manual future
HAL		✓		✓	
Harmony					Thread object
Heraklit		✓			
HoME	✓				
Hybrid		✓			
Karos		✓			
LO		✓			
MeldC		✓			
Mentat		✓		✓	
Meyer's proposal		✓			
MPC++		✓	✓		
Multiprocessor-smalltalk	✓				
Obliq	✓				Return value
Orca	✓				
Parallel computing action		✓	✓		
Parallel Object-oriented Fortran		✓			
PO		✓	✓		
POOL				✓	
Presto					Thread object
Procol		✓			
pSather		✓	✓		Queue
PVM++					Thread object
Python					Thread/fork, library
QPC++		✓		✓	Wait by necessity
Rosette		✓		✓	
SAM		✓		✓	
Scoop					Thread object
Smalltalk	✓				
SR		✓		✓	
Tool		✓			
Trellis/Owl					Thread object
Ubik		✓		✓	
UC++		✓			

Table 2: COOLs in Cobegin, Par and equivalents category

Cools	Cobegin	Par statement	Activity st	Misc. features
ABCL/x	✓			
COOL/Stanford				Wait for statement
Comp. C++	✓			
Conc. aggregates	✓			
DOWL			✓	
Guide	✓			
LO				Combination Join
Micro C++				Block D thread boundary
Proof	✓			
pSather		✓	✓	
Rosette	✓			
Scheduling predicates	✓			
SOS	✓			
Spar	✓			
SR	✓			Co-statement
Trellis/Owl	✓		✓	

Many COOLs make this dependence explicit. They introduce so-called futures that are special variables with the following characteristic: after a value has been written to the future, the future behaves like a plain variable. An activity that tries to read from an un-initialized future is blocked until another activity writes to the future.

Asynchronous calls and futures restrict parallel performance due to limited fan-out and may break modularity unless encapsulation is preserved by callee-side coordination.

Post-processing: Early return (also known as post-processing) is dual to asynchronous method calls. Whereas in the case of the asynchronous method call parallelism is introduced at the point of the method call, post processing results in initiation of parallelism at the point of return. With post-processing the called method can return a result but continue to work.

Cobegin, par and equivalents: Table 2 deals the syntaxes of Cobegin, Par and equivalents.

Cobegin: The cobegin statement (Kessels, 1977; Papathomas, 1989; Briot *et al*., 1998) is a structured form of initiating parallelism in a language. In contrast to fork-join and their equivalents, this control structure obeys the single-entry single-exit paradigm. The execution

Cobegin StmtList1 |…..| StmtListn end

of creates n concurrent activities, each of which executes the corresponding list of statements. The essential difference to fork-join is that the original thread only continues when all n threads themselves have terminated. Whereas the join statement was optional and several join statements could refer to a single fork, the cobegin statement syntactically enforces a synchronization.

Par and equivalents: The par statement is similar to the cobegin statement in its characteristic that the original activity is blocked until all activities that are spawned inside the par statement are terminated.

par StmtList end

The par statement itself does not introduce any parallelism but is solely used to coordinate concurrency. Only if StmtList itself initiates concurrency does the above mentioned synchronization take place. A cobegin can be equivalently expressed by means of par and fork:

Activity set: The programmer can explicitly add activities to an activity set and then wait for the completion of all those activities. Although the effect is similar to the par statement, it no longer provides the ease of understanding of potential concurrency. Whereas the par statements narrow the concurrent activities to a couple of program lines, the activity set can be modified anywhere in its scope.

Forall, aggregate and equivalents: This section covers constructs such as Forall, aggregate and equivalents that may start many concurrent activities at once. The readers may find this information related to COOLs in Table 3. These activities are bound to objects or specific data structures because each new activity is supposed to only work on a particular object or data element of a given data structure.

Forall: Various forms of the cobegin statement found their way into parallel languages. Most notably, the forall, doall and doacross forms. Several instances of StmtList are executed concurrently, one for each element in the range.

forall I:[range] do StmtList(I) end

The forall may break modularity unless encapsulation is preserved by callee-side coordination or by strictly

Table 3: COOLs in forall, aggregate and equivalents category

COOL	Forall	Aggregate	Multicast	Miscellaneous
Actorsace		✓		
A-NETL			✓	
Arche		✓		
Blaze-2	✓			
Braid				Data parallel
C**				Data parallel
CHARM++				Cluster aggregate
Comp. C++	✓			
Conc. aggregates		✓		
dpSather				Data parallel
EPEE				Cluster aggregate
Fragmented objects, FOG/C++			✓	
HPJava	✓			
Modula-3*	✓			
NAM				Data parallel
parallel C++				Data parallel
Procol			✓	Via type
QPC++		✓		Processor set
Spar	✓			
SR				Array of process: strip, co-stmt: quantifier
Titanium	✓			

Table 4: COOLs in autonomous code category

COOLs	Life routine	Autonomous routine	Self start	Miscellaneous
Arche	✓		✓	
Atom		✓		
Beta		✓		Separate start
C++//	✓		✓	
CEiffel		✓	✓	
COB	✓		✓	
Conc. class eiffel	✓			Separate start
Correlate		✓	✓	
Distr. eiffel				Process: dynamic
DoPVM				Process: static
Dragoon		✓	✓	
Eiffel//	✓		✓	
Emerald		✓	✓	
Guide				Process: static
IceT				Process
Java		✓		Separate start
Java//, ProActive PDC	✓		✓	
Mediators	✓		✓	
Mentat	✓		✓	
Micro C++	✓	✓	✓	
Moose		✓	✓	
Panda		✓	✓	
POOL	✓		✓	
Proof		✓	✓	
Guide (process, static)				Process: static
QPC++	✓		✓	
SR	✓	✓	✓	
Titanium				Process

confining concurrency to individual data elements (one per instance of the forall), i.e., by avoiding any data dependence within the forall. Here as cobegin helps in determining these activities which could be executing concurrently, the forall statements further reduce complexity by restricting what the activities can do to a single statement list.

Aggregate, multicast and cluster aggregates: Aggregate languages offer a mechanism to group together several objects and then call a particular member function for all objects of this aggregate. Similar to the forall where a data structuring concept of the language, i.e., the array, is used to express that an operation must be performed on all elements (Wegner, 1987).

Languages that are based on explicit message passing sometimes define aggregates implicitly by defining lists of recipients or by linking several recipients to a single communication channel. This is called multicast message passing.

Autonomous code: This section covers constructs that start one new concurrent activity at a time. This activity is bound to an object, a specific data structure of the language, or to a code sequence. The COOLs in Autonomous code category is given Table 4.

Process: Fork-join, cobegin, forall and aggregates initiate parallelism at arbitrary points of an otherwise sequential program. Instead, process declarations make parallelism explicit and are targeted towards coarse grain parallelism where a few clearly identifiable tasks exist. Process P is Procedure-Body end Processes do not express parallelism within the object-oriented program; they are a language concept that exists on top of an otherwise object-oriented language.

Autonomous routine: An extension of process declarations is to combine them with object declarations. When an object is created, an additional activity is spawned that executes a specific member function, called autonomous routine. Whereas in some languages (Wegner, 1987; Cardelli and Wegner, 1985; Korson and McGregor, 1990) autonomous routines are started automatically upon object creation, other languages require an explicit start of that method.

COORDINATING CONCURRENCY

This section surveys coordination constructs found in COOLs. The main distinction is made with respect to the goal of callee-side coordination (Meyer, 1988; Goss and Hartmanis, 1983).

Activity-centered coordination: Mechanisms in this group do not fulfill the goal of callee-side coordination though some allow fulfilling it manually. The COOLs in activity centered coordination category is given in Table 5.

Synchronization by termination: Most of the mechanisms for initiating parallelism provide a simple way of synchronization. Whereas in practice fork-join and cobegin programs rely on other means of coordination, data-parallel and aggregate programming are solely based on synchronization by termination.

Semaphore, mutex and lock: The semaphore is another basic concept of organizing concurrent access to shared data. A semaphore is a non-negative integer variable with two atomic operations. A critical section of the code, i.e., a section that operates on shared data, must be enclosed by a pair of these operations. The other operation, called V or signal, increases the variable atomically. When the value of the variable is greater than one, more than one activity can pass. Blocked activities are queued. Mutex and lock are special types of semaphores that allow exactly one activity to enter a critical section. Both mutex and lock can easily be implemented with semaphore operations.

Piggy-backed synchronization: In pure message passing languages, concurrently executing activities are often synchronized by blocking communication commands. The receive statement waits until a message m arrives from a specific sender s. Thus, the synchronization is piggy-backed on top of the communication. Here is the syntax:

receive m [from s]

Boundary coordination: We distinguish three general forms of boundary coordination mechanisms. The distinction is based on the division of responsibility between the run-time system and the object. The basic question is where the coordination code is placed. The

Table 5: COOLs in activity centered coordination category

COOLs	Termination	Semaphore	Mutex	Lock	Misc. features
Amber				✓	Barrier
Atom					Enable set, instead of 'become' like guarding conditions
Beta		✓			
Blaze-2	✓			✓	
Braid	✓				
C**	✓				
Comp. C++					Coordination future
Conc. Aggregate				reader/wr iter lock	
Conc. smalltalk		✓			
cooC		✓			
COOL/Chorus		✓			
CST		✓			
Distr. C++					Coordination future
Distr. eiffel		✓		✓	
Distr. smalltalks		✓			
DoPVM				✓	
DOWL				✓	
dpSather	✓				
EPEE	✓				
ES-Kit				✓	
Harmony		✓			
HoME		✓			
HPJava	✓				Piggy-backed Sync.

Table 5: Continue

COOLs	Termination	Semaphore	Mutex	Lock	Misc. features
IceT					Piggy-backed Sync
Distr. C++	✓		✓		
Karos	✓				
LO	✓				
MeldC		✓	✓		
Modula-3*	✓				
MPC++ (mutex)			✓		
Multiprocessor-smalltalk		✓			
NAM	✓				
Obliq			✓	✓	
Panda		✓			
parallel C++	✓				
PO		✓			
Presto			✓	✓	Coordination future
Proof				✓	
pSather				✓	
PVM++		✓		✓	
Python		✓	✓	✓	
QPC++		✓			
Scoop					Piggy-backed Sync
Smalltalk		✓			
Spar	✓		✓		
SR		✓			
Titanium	✓				Piggy-backed Sync
Trellis/Owl				✓	

Table 6: COOLs in boundary coordination category

COOLs	Monitor	Condition variables	Conditional wait	Misc. features
Amber	✓			
A-NETL	✓			
A'UM	✓			
CHARM++	✓			
Conc. smalltalk	✓			
COOL/NTT	✓			
COOL/Stanford		✓		
CST-MIT	✓			
Emerald	✓			
ESP	✓			
Fleng++	✓			
Fragmented objects, FOG/C++	✓			
Heraklit			✓	
Mentat	✓			Sequential and persistent objects
Micro C++	✓			
Obliq		✓		
Orca	✓			
Oz, Perdio	✓			
Panda	✓			
Presto		✓		
SAM	✓			
Tool	✓			
UC++	✓			

COOLs in Boundary Coordination category is given in Table 6.

Implicit control: COOLs based on boundary coordination with implicit control define for all classes whether and which of concurrently invoked methods to execute. The programmer does not write explicit concurrency coordination code. The run-time system is responsible for proper coordination.

Monitor: Implicit control is an instance of the monitor concept (Yaoqing and Kwong, 1993; Wyatt *et al*., 1992). In object oriented terminology, a monitor is an object that has internal variables to implement its state and offers methods that operate on that state under a mutual exclusion regime.

Condition variables: In this monitor extension, an activity that has entered a monitor can block inside of the monitor

Table 7: COOLs in isolated hand shake category

COOLs	Method guard	Enable set	Life routine	Serialized method	MISC.
Acore					Un serialized
ACT++		✓			Called: behavior set
ASK				✓	
Arche		✓			Reader/writer Protocol
Blaze-2				✓	Also: lock
C++//			✓		1st class methods
CEiffel	✓				Method compatibility
CLIX	✓				
COB			✓		
Comp. C++				✓	Also: coordination future
Conc. aggregate					Un serialized, reader/writer Protocol
Conc. class eiffel			✓		
cooC				✓	Also: semaphore
COOL/Stanford				✓	Also: condition variable
Correlate	✓				
Demeter	✓			✓	
Distr. Eiffel	✓				Reader/writer Protocol
Distr. smalltalk-process	✓			✓	Also: semaphore
Dragoon	✓				Counter
Eiffel//			✓		1st class methods
Guide	✓				Counter
HAL	✓				
Java				✓	Also: mutex
Java//, ProActive PDC			✓		1st class methods
Mediators (life routine,)	✓		✓		Receive, counter
Mentat			✓		Receive
Meyer's proposal	✓				
Micro C++			✓		Receive
Moose	✓				
Obliq				✓	Also: mutex, lock
Orca	✓				
Parallel computing action	✓				
PO	✓				
POOL			✓		Receive
Procol	✓				Path expression
Proof	✓				
QPC++			✓		Receive
Rosette	✓	✓		✓	
Scheduling predicates	✓				Counter
SOS	✓				Counter

at the condition variable by calling cond var.wait. While it is blocked, another call can proceed. The first activity blocks until the other activity calls cond var.signal. Since the monitor's one-activity-at-a-time principle is obeyed it must be specified what happens after a signal call, when conceptually at least two activities are ready to proceed.

Conditional wait: This variant of condition variables has been introduced by Kessels (1977) to improve the conditional synchronization in monitors. Kessels (1977) proposed to isolate the conditions syntactically instead of spreading wait and signal over the class.

condition identifier : cond-expr;

Although some of the problems are solved, it cannot be specified which of a collection of blocking activities is continued when the condition holds. It is much easier to identify the relevant conditions in the code, but the programmer can still be tricked into deadlocks.

Handshake control and isolated handshake control: Boundary coordination with handshake control divides the responsibility for coordination between the object's implementation and the run-time system (or the handler of message queues) (Borning, 1986). In general there is code in the class that has the sole purpose of specifying the concurrency coordination, i.e., the object's dynamic interface. Handshake control mechanisms fulfill the goal of callee-side coordination. The COOLs in Isolated hand Shake category is given in Table 7.

Method guard: With method guards, proceed-criteria can be expressed similar to preconditions. Before a guarded method is executed its condition is evaluated. If it holds,

Table 8: COOLs in reflective control category

COOLs	Reflective control	Misc. features
ABCL/R2	✓	
ABCL/R3	✓	
DROL		Protocol object, 1-activity/time
HAL	✓	
MeldC		Shadow object, intra-object conc.

the method is invoked, otherwise the call is delayed. The condition can be an expression over all instance variables of the object.

Enable set: One of the remaining problems with behavior abstractions is that each method has to perform a possibly complex analysis to determine the new behavior in the transition phase. When the sets of possible states change in subclasses, this analysis must be re-worked in otherwise unaffected methods.

Life routine: It specifies coordination procedurally. The constructs discussed below use the message terminology instead of understanding messages as method calls.

Reflective control: COOLs (Castagna, 1995) based on boundary coordination with reflective control keep class implementations free of coordination code. In contrast to implicit control, where there is no explicit coordination code, with reflective control the programmer can explicitly formulate the coordination constraints in meta-classes. The COOLs in Reflective Control category is given in Table 8.

LOCALITY

A COOL that is implemented on a parallel computer faces the mapping problem, i.e., the COOL must provide for a mapping of objects and activities to memory modules and processing elements. On parallel computers, the notion of locality is essential for achieving appropriate run-time performance since access times to memory are more non-uniform than they are for single processor computers. The COOLs in Locality category is given in Table 9.

Meta-level locality: COOLs with reflective concurrency control usually extend the reflective approach to the mapping problem. Distribution is completely transparent, i.e., given an object reference, it is impossible to decide statically whether the object is stored locally or on a remote processor. The programmer is in charge to implement appropriate mapping strategies procedurally in the meta-class. No locality-enhancing work need be done by the run-time system and compiler. When a new object is created, the meta-object assigned to the class is consulted first. Since every method invocation goes through the meta-object first, it is possible to implement object migration.

External locality: COOLs with external locality have object placement that is beyond the scope of the language. The user is in charge of the mapping, by manually placing objects on various machines and registering their location with a name server. The user then binds variables to possibly remote objects by asking the name server for a reference.

Internal locality: The programmer can optionally specify the processor that must be used to store an object or to execute a thread. If the specification is omitted, an automatic default mapping strategy is applied. Often, the new statement is augmented with an optional processor number or the statements to initiate concurrency have an additional syntactic feature to guide thread creation.

Virtual topology/scope locality: The difference between internal locality and virtual topology/scope locality is that the programmer has an abstract model of the parallel machine in mind. Objects and threads are mapped onto this model, e.g. by means of abstract processor numbers, instead of mapping them directly to the hardware. The abstract model is automatically mapped to the underlying machine topology. The visibility rules reflect locality, i.e., only elements that are declared in the same segment are stored locally and can be accessed directly; access to other elements requires additional syntactic overhead, reflecting the cost of non-locality. Most COOLs in this category do not offer object migration. There are no dynamically changing virtual topologies.

Group locality: In the approaches discussed so far, the mapping is specified procedurally or declaratively. In contrast, COOLs with group locality specify characteristics that shall be fulfilled by all potential mappings. It can be expressed that certain objects should be kept together by the automatic mapping. Often, objects are 'attached' to other objects. The programmer explicitly

Table 9: COOLs in locality category

COOLs	Meta level	External	Internal	Virtual topology/scope	Group	Misc. features
ABCL/1			✓			
ABCL/f			✓			
ABCL/R2	✓					
ABCL/R3	✓					
Amber			✓		✓	
A-NETL					✓	
Beta			✓		✓	
Braid						Data //, arrays
Comp. C++				✓		
Conc. aggregates					✓	
Conc. class eiffel	✓					
COOL/ Choms					✓	
COOL/NIT		✓				
COOL/Stanford			✓		✓	
Correlate	✓					
Distrib. C++				✓		
Distrib. Eiffil				✓		
DOWL			✓		✓	
DpSather						Data //
DROL	✓					
Emerald			✓		✓	
EPEE				✓		
Guide			✓			
HP Java						Data //, at, on
Java//, ProActive PDC	✓					
JavaParty	✓		✓			
Java/RMI		✓				
MeldC	✓					
Mentat			✓		✓	
MPC++				✓		
NAM				✓		
Obliq		✓				
Panda			✓			
Parallel C++				✓		
POOL			✓			
Procol		✓				
pSather						Zones
Rosette			✓			
SR				✓		
Titanium						Zones
UC++			✓			

forms networks of objects that belong together. The run-time system then maps the networks to the underlying topology (Yonezawa, 1990).

CONCLUSIONS

The combination of parallel and object-oriented paradigms in the design of COOLs raises various difficulties, since these paradigms have some contradictory issues. The aspect of concurrency coordination is well researched: Enable sets, standard life routines and reflective control solve most of the concurrency coordination problems. There are two major aspects that need more attention. First, to specify how to map objects and activities for locality there are almost no mechanisms that would blend with object or class-based programming. Second, the area lacks quantitative and empirical data. The COOLs are not used for enough application code, almost no performance data is published for quantitative evaluations and comparisons and there are no comparative figures about programming error probability and maintenance time.

REFERENCES

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

America, P., 1987. Inheritance and subtyping in a parallel object-oriented language. ECOOP (Lecture Notes in Computer Science, 276: 234-242.

Bal, H.E., J.S. Steiner and A.S. Tanenbaum, 1989. Programming languages for distributed computing systems. ACM Computing Surveys, 21: 261-322.

Bal, H.E., 1991. A comparative study of five parallel programming languages. Spring Conference on Open Distributed Systems, EurOpen, pp: 209-228.

Borning, A.H., 1986. Classes versus prototypes in object-oriented languages. ACM/IEEE Fall Joint Computer Conference.

Briot, J.P., R. Guerraoui and K.P. L¨ohr, 1998. Concurrency and distribution in object-oriented programming. ACM Computing Surveys, 30: 291-329.

Cardelli, L. and P. Wegner, 1985. On understanding types, data abstractions and polymorphism. ACM Computing Surveys, 17: 471-522.

Castagna, G., 1995. Covariance and contravariance: Conflict without a cause. ACM TOPLAS., 17: 431-447.

Goos, G. and J. Hartmanis, 1983. The Programming Language Ada Reference Manual, ANSI. MIL-STD-1815A-1983.

Kessels, J.L.W., 1977. An alternative to event queues for synchronization in monitors. Communications of the ACM., 20: 500-503.

Korson, T. and J.D. McGregor, 1990. Understanding object-oriented: A unifying paradigm. Communications of the ACM., 33: 40-60.

Meyer, B., 1988. Object-Oriented Software Construction. Prentice Hall.

Papathomas, M., 1989. Concurrency Issues in Object-oriented Programming Languages. Object Oriented Development, Tsichritzis, D. (Ed.). University of Geneva, Switzerland, pp: 207-245.

Turcotte, L.H., 1993. A survey of software environments for exploiting network computing resources. Technical Report, Mississippi State University.

Wegner, P., 1987. Dimensions of object based language design. OOPSLA., pp: 168-182.

Wyatt, B., K. Kavi and S. Hufnagel, 1992. Parallelism in object-oriented languages: A survey. IEEE Computer, 11: 56-66.

Yaoqing, G. and Y.C. Kwong, 1993. A survey of implementations of concurrent, parallel and distributed Smalltalk. ACM SIGPLAN Notices, 28: 29-35.

Yonezawa, A., 1990. ABCL: An Object-Oriented Concurrent System-Theory, Language, Programming, Implementation and Application. MIT Press.