# A Survey of Concurrent Object-Oriented Languages

Michael Philippsen
Computer Science Department, University of Karlsruhe, Germany

During the last decade object-oriented programming has grown from marginal influence into widespread acceptance. During the same period, progress in hardware and networking has changed the computing environment from sequential to parallel. Multi-processor workstations and clusters are now quite common.

Unnumbered proposals have been made to combine both developments. Always the prime objective has been to provide the advantages of object-oriented software design at the increased power of parallel machines. However, combining both concepts has proven to be notoriously difficult. Depending on the approach, often key characteristics of either the object-oriented paradigm or key performance factors of parallelism are sacrificed, resulting in unsatisfactory languages.

This survey first recapitulates well-known characteristics of both the object-oriented paradigm and parallel programming, and then marks out the design space of possible combinations by identifying various interdependencies of key concepts. The design space is then filled with data points: For 111 proposed languages we provide brief characteristics and feature tables. Feature tables, the comprehensive bibliography, and web-addresses might help identifying open questions and preventing re-inventions.

Categories and Subject Descriptors: C.2.4 [**Computer Communication Networks**]: Distributed Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 [**Software Engineering**]: Tools and Techniques; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.1.2 [**Computation by Abstract Devices**]: Models of Computation—*Parallelism and Concurrency*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs

General Terms: Programming Languages, Object-Orientation, Parallelism, Concurrency

Address: University of Karlsruhe, Institute for Program Structures and Data Organization, Am Fasanengarten 5, D-76128 Karlsruhe, Germany; email: phlipp@acm.org.
This work was partially supported by ICSI, International Computer Science Institute, Berkeley.

## 1. INTRODUCTION

The prime objective for COOL (Concurrent Object-Oriented Language) designs is to combine the advantages of object-oriented design with the increased power of parallel machines. This article surveys proposed COOL designs and shows that combining both concepts has proven to be notoriously difficult. Often key characteristics of the object-oriented paradigm or key performance factors of parallelism are sacrificed.

In section 2 we illustrate problems that commonly occur when concurrency is introduced into object-oriented languages. We derive remedies that can either be applied by a COOL programmer as a programming style, or should be goals of COOL designs. The issues identified in this section are discussed in detail in the following sections.

Sections 3 and 4 present COOL mechanisms for initiation of parallel activities and for coordination of concurrency (often called synchronization). Section 5 covers approaches to achieve adequate performance on the underlying parallel platforms by enhancing locality of objects and activities.

Sections 3 to 5 mainly address two types of readers. For tutorial readers, existing COOL ideas are presented and organized to give a comprehensive overview of the field. For language designers the dimensions spanning the design space of concurrent object-oriented programming languages are discussed. We analyze advantages, disadvantages, and interdependencies of certain language features.

Throughout the survey, we represent language features with graphical symbols. By labeling each language with a pictograph in appendix A, the reader can visually understand and compare language designs – for example to see which of the common problems remain unsolved in which COOLs. Instead of adding citations wherever a COOL is mentioned, we postpone all citations to this appendix.

The languages discussed in this survey are primarily aimed at writing single programs that solve a problem with explicit parallelism. We do not consider programming environments that are oriented towards distributed programming, i.e. that offer an interface definition language (IDL) to allow several programs to cooperate on shared objects.

### 1.1 Related Work

[Andrews and Schneider 1983] survey basic concepts of concurrent programming; [Bal et al. 1989] discuss some programming languages for distributed computing. There are several survey articles on aspects of concurrent object-oriented languages. [Bal 1991] discusses five languages. [Nuttal 1994] discusses systems that provide process or object migration. [Cheng 1993] describes a collection of languages and tools, some of which are object-oriented. [Yaoqing and Kwong 1993] survey parallel and distributed Smalltalks. [Wyatt et al. 1992] study several languages and discuss whether the parallelism is appropriately integrated into the languages. [Karaorman and Bruno 1993a] elaborate on the design space of parallel object-oriented programming. Papathomas [1989, 1992] gives a first classification of COOLs. He focuses mainly on the combination of concurrency with objects (see part of section 4) and does not classify the broad number of languages we look at. Neither does he take more machine-oriented details into account, e.g., the way objects or activities are mapped to the underlying parallel hardware. Collections are due to [Turcotte 1993] and [Philippsen 1995b].

[Briot et al. 1998] do not consider as many languages, have a coarse classification, and ignore the locality issues.

### 1.2 Basics of Object-Orientation

Let us briefly recapitulate the concepts of object-oriented languages to introduce the terminology used in this article. Our terminology is based on [Wegner 1987; Cardelli and Wegner 1985; Korson and McGregor 1990; Meyer 1988; Pressmann 1987].

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. All objects of a class have the same interface, they offer the same set of methods, implement the same behavior, and therefore belong to the same type. The difference between the terms **class** and **type** is discussed below in the context of inheritance.[1]

Languages with objects, but without the notion of classes are called **object-based** languages. 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 [Borning 1986]. In class-based programming languages, classes hide information regarding their internal details behind a well defined interface and hence support a modular system design [Parnas 1972]. A given class should be easy to replace by an alternative implementation that offers the same interface. An extension of the class-based approach are classes with type arguments. These **generic classes** or **template classes** ease code reuse, since a useful abstract data type needs to be implemented only once. By specifying the type argument, the generic class turns into a concrete class which can then be used to instantiate objects.

**Inheritance** is the essential feature that turns class-based languages into **object-oriented** languages. The general concept is reuse in a broad sense, namely that new or more specific implementations can be made from existing or more general implementations. There are three general types of inheritance; most object-oriented languages have only one type.

| | class | object |
|---|---|---|
| implementation | implementation inheritance | delegation |
| interface | type-based inheritance | |

It is called **implementation inheritance** (or simply inheritance) if the implementation of a subclass is defined by extending an existing class with a new feature or by specializing the implementation of a method.

Class-based inheritance can also be defined **based on types**, i.e., the class signatures alone. Note, that some object-oriented languages offer both implementation inheritance and type based inheritance. A class is below another

[1] The difference between classes and Ada's packages [ANSI 1983] is that classes determine types of the language. Objects of a class instantiate this type. Packages cannot be used to instantiate objects, but only encapsulate types.

class in the **type hierarchy**, if the class offers at least the same methods. To be more exact: For all method calls (for all types of parameters and return values) that are well defined for the upper type, the lower type provides some implementation. Depending on the particular language design, different sub-type relations (co-variance and contra-variance) are required for parameters and arguments [Castagna 1995]. In contrast to implementation inheritance it is not necessary that a class and its subclass have a common code base.

An alternative to class-based inheritance is **object-based inheritance**, often called **delegation** [Snyder 1986]; see the right column of the above table. When a method is invoked that is not explicitly provided by the object's implementation, the object delegates the call to another object that then is bound to "self". This second object then invokes the corresponding method unless it again needs to delegate the call. Delegation can also be used for object-based and prototype-based languages.

An object-oriented language is said to offer **multiple inheritance** if a new class can inherit from the implementations of more than one (unrelated) ancestors or if a new class can be in the type hierarchy below two different types, i.e., the new class offers an interface that is a combination of both interfaces of the parent types. The semantics of the language has to define how various sorts of conflicts are resolved. It is a form of multiple inheritance if an object uses several other objects to delegate method calls.

In addition to the software quality features gained by class-based languages (i.e., support for modular design and reuse of generic classes), it is an additional benefit that code can easily be extended and reused in subclasses. For full extensibility, however, the following two additional characteristics have to be offered by the language: **Polymorphism** and **dynamic binding**. A polymorphic reference cannot only refer to objects of a particular declared **static type**, A, but as well to objects of any subtype of A. The concrete type of an object that is actually stored in a variable at run-time is called **dynamic type**. Only if polymorphism is offered can an object of a newly defined subtype substitute objects of the ancestor type in old code. Polymorphism and dynamic binding are two sides of the same coin. While polymorphism allows a variable to hold objects of a type or its subtypes, with dynamic binding method invocations are mapped at run-time to the implementation that belongs to the dynamic type.

## 2. COOL PROGRAMMING PROBLEMS

When concurrency is added to an object-oriented language certain common problems are caused. Most of these problems can be avoided by appropriate programming style or language design.

### 2.1 Parallel Performance

The desire for performance originally drove the development of COOLs. Three aspects need to be considered.

*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 $\lceil log(p) \rceil$ 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**, i.e., COOLs should offer ways to create more than one new activity at a time. In appendix A the languages that offer such constructs are represented with a star that has one of its upper arms shaded. Section 3 presents ways to initiate concurrency in COOLs and discusses each of the arms of the star.

*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.

There are reasons for prohibiting intra-object concurrency. First, without intra-object concurrency it is easier to reason about the correctness of a class implementation, see section 2.2. Second, the programmer does not need to implement any form of concurrency coordination to express whether and which methods can correctly be executed concurrently. Hence, most types of inheritance anomalies can be avoided, see section 2.3. Section 4 surveys and compares mechanisms to express concurrency coordination.

*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. Section 5 discusses **ways to achieve locality** in object-oriented languages. Interestingly, more than half of the languages we look at do not consider this problem at all. Locality issues are not depicted in appendix A.

### 2.2 Broken Encapsulation

In *sequential* languages, a modular design [Parnas 1972] divides a system into smaller structures and reduces complexity significantly since correctness considerations can often be limited to boundaries of modules that then can be tested in isolation. This kind of encapsulation is a central paradigm of object-orientation and a major software quality factor. Encapsulation is needed for COOLs as well.

To reason about the correctness of *sequential* object-oriented programs Meyer's principle of "design by contract" [Meyer 1988; Meyer 1992] can be applied. A class $\mathsf{C}$ is *locally correct* if

—after instantiation of a new object of $\mathsf{C}$, the class invariant $Inv_{\mathsf{C}}$ holds and

—after execution of a method $\mathsf{m}$ of class $\mathsf{C}$ both the class invariant $Inv_{\mathsf{C}}$ and the post-condition $Post_{\mathsf{m},\mathsf{C}}$ of that method hold, provided that both the invariant and the pre-condition $Pre_{\mathsf{m},\mathsf{C}}$ were fulfilled at the time of the invocation. Formally:

$$(I) \qquad Inv_{\mathsf{C}} \wedge Pre_{\mathsf{m},\mathsf{C}} \xrightarrow{\mathsf{m}} Inv_{\mathsf{C}} \wedge Post_{\mathsf{m},\mathsf{C}}$$

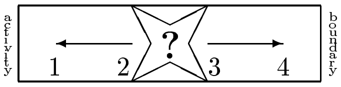
A caller who knows the interface or the specification of a method can call it without knowledge of its implementation details. Hence, the implementation can be changed without affecting any callers.

Meyer's definition of local correctness cannot easily be used for concurrent programming. Since the implication (I) does not say anything about whether the invariant holds *during* the execution of a method it cannot be concluded that the class invariant holds at all times. There might be interleavings of method invocations that result in a broken invariant and thus incorrect code. However, there is a way to apply Meyer's concept of local correctness to COOL programming while allowing restricted intra-object concurrency.

The idea of **encapsulation by callee-side coordination** is to implement concurrency coordination at the side of the callee, i.e., in the class that is accessed concurrently.[2] Language constructs that achieve callee-side coordination by design are called **boundary coordination** mechanisms. As we discuss in section 4.2, with such mechanisms the class implementation can assure that methods will only be executed concurrently if their interleaving does not affect correctness. Interfering method invocations will be delayed. With callee-side coordination it is possible to reason about the correctness of a class implementation based on local information since all coordination code is part of the class implementation.

In contrast, in COOLs with **activity-centered coordination** intra-object concurrency is available, but *the caller* is responsible for coordinating concurrency. The activities themselves must make sure that access to shared data is properly coordinated for avoiding race-conditions. This may break encapsulation for two reasons. First, the caller must know the implementation details of the invoked method to be convinced that the method can be executed concurrently without harmful interference. Second, changing the implementation of a (called) method requires careful analysis of all code positions that call this method since the new implementation might require coordination constraints not yet implemented in all the callers. Both problems are especially painful for COOLs because of their support for code reuse and dynamic binding. New subclasses may be provided later that will no longer work correctly with older callers.

Therefore, if a particular COOL does only offer activity-centered coordination, the programmer should still try to code according to the principle of callee-side coordination.




In appendix A, languages with only boundary coordination mechanisms are depicted with a star on the right (at positions 3 or 4) of the slide-bar. For boundary coordination mechanisms, the interior of the star is refined and reveals additional details, see below. Languages that have activity-centered coordination mechanisms are depicted on the left (positions 1 or 2). Some COOLs offer both activity-centered and boundary coordination. In these cases, the refined stars is on the "activity side" of the slide-bar (to take advantage of the redundancy).

### 2.3 Inheritance Anomalies

Even if encapsulation is preserved, i.e., if concurrency coordination is always implemented on the side of the callee there are reuse problems.

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. 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. The other methods must be redefined in the sub- or superclass as well, which degrades maintainability and prevents

[2]We will see later that encapsulation can also be preserved if concurrency is confined within objects, see for example section 3.2.3.

reuse. But even if the coordination code is better isolated, it is often necessary to refine the complete coordination code for all methods instead of allowing local extensions of parts of the coordination code. These and other related difficulties of combining concurrency coordination and inheritance are collectively called **Inheritance Anomaly**. For a more detailed discussion and for examples see [Matsuoka and Yonezawa 1993; Frølund 1996; Kafura and Lavender 1996; McHale 1994; Kafura and Lee 1989; Briot and Yonezawa 1987; America 1987a; Thomas 1992; Papathomas 1989; Tomlinson and Singh 1989].

As an essence of the work on inheritance anomalies this survey considers four major design goals for COOLs, two of which are covered in section 2.4.

The effects of inheritance anomaly can be reduced with **isolated coordination code**, i.e., if concurrency coordination code is isolated from code that implements class functionality. Only with isolated coordination code, functionality and concurrency coordination can be inherited separately.

But even with isolated coordination code some inheritance anomalies still occur. If the concurrency coordination is expressed in a centralized way, for example with a single construct, it is often necessary to re-program the complete coordination code in a subclass even if that subclass has just slightly different coordination constraints. Therefore, it is desirable to have **separable coordination code**, where portions of the coordination code can be refined while other portions are reused.

A detailed discussion of the language mechanisms and their treatment of inheritance anomalies is given in section 4. This discussions can guide the programmer to achieve cleaner inheritance by manually isolating and separating coordination code if a particular COOL does not offer supporting constructs.

For COOLs with isolated coordination code, the star is at position 4 of the slidebar in appendix A. Or it is at position 2 if the COOL also has activity-centered coordination mechanisms. If the coordination mechanism is not isolated, the star is at position 3 (or 1).

### 2.4 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.

A difference between pre-conditions and proceed-criteria is that the former are not meant to be actually evaluated during program execution. Pre-conditions are helpful to find errors during the implementation phase but they do not contribute to the semantics of the program. In contrast, proceed-criteria must be checked at each method invocation. Moreover, pre-conditions and proceed-criteria are different with respect to inheritance. Pre- and post-conditions are determined by the algorithms and the abstract data type offered by the class or subclass. In contrast, proceed-criteria represent coordination constraints that are caused by the chosen implementation. They might change if a different implementation requires other concurrency constraints. It is because of these differences that inheritance of proceed-criteria is an issue in COOLs. Two general types of proceed-criteria can be identified in COOL programming.

**State proceed-criteria** express that a method call can proceed or must be delayed because of a condition of the internal state of the object.

**History proceed-criteria** express that a method call can proceed or must be delayed because of the history of earlier calls processed by the object.

Both types of proceed-criteria can be implemented manually by means of additional instance variable and additional code. However, it has been shown by [Matsuoka and Yonezawa 1993] that this will cause inheritance anomalies. The same is true if only one type of proceed-criteria is supported by a COOL. To implement the other type, again additional instance variables and code are needed that cause inheritance anomalies. COOLs should therefore provide language elements to clearly express both types of proceed-criteria.

Section 4 discusses the expressiveness of coordination mechanisms in detail.

### 2.5 Language versus Library

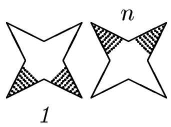
Some COOLs extend existing object-oriented languages by adding concurrency in a library. Special classes are offered that add for example semaphore operations. [Buhr 1995] points out reasons why such library extensions can in general not be implemented correctly. The basic insight is that the compiler does not know about the special concurrent semantics of the the added libraries. Hence, it cannot be guaranteed that compiler optimizations do not interfere with these semantics. Let a be an object that inherits both a **lock** and an **unlock** method from a library class.

```
a.lock();
a.method();
a.unlock();
```

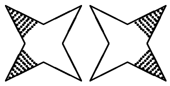
Inside of the critical section between the call of **lock** and the call of **unlock** the above code executes foo which presumably causes some race-conditions if not executed in isolation. However, a standard compiler that does not know about the special concurrent semantics of **lock** and **unlock** might change the evaluation order if it can prove the absence of dependences. The compiler will not change the relative order of **lock** and **unlock** but might move the call of foo out of the critical section, because in general there are no dependences between the instance variables used to implement the lock and the ones used in foo. This "optimization" destroys the correctness of the code and causes erratic behavior that is almost impossible to debug. Buhr points out several related problems all of which can be explained by the unawareness of a standard compiler of the additional concurrent semantics.

## 3. 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. The mechanisms are categorized into five groups each of which is discussed in a subsection below. Each subsection is labeled with a star-shaped pictograph that is used in appendix A for a visual comparison of existing COOLs.

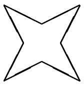



We structure the discussion along two axes. One axis deals with the number of parallel activities that can be spawned with a single language construct. One of the lower two arms of the star is shaded if a language offers constructs to spawn a single new activity. If one of the upper arms is shaded more than one parallel activity can be spawned at a time.



The second axis reflects different understanding of parallelism. The left half represents a thread-centered understanding: A new activity is created and this activity is not coupled to any of the objects in the program. The right half represents a more object-centered understanding: Either the parallel activity is intended to operate only on a particular object or it is syntactically bound to a special class or object.

## 3.1 Automatic Parallelization



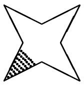
The sequential representation provided by the programmer is automatically converted into a parallel form. Conceptually, automatic parallelization fits well to object-oriented programming languages since it does not visibly interfere with existing language characteristics. We represent automatic parallelization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

Although significant progress has been made on well defined sub-problems (array based data dependence analysis [Banerjee 1988; Wolfe 1989], pointer or alias analysis [Banning 1979; Choi et al. 1993; Kooper and Kennedy 1989; Landi et al. 1993], and other relevant techniques [Bacon et al. 1994]), automatic parallelization cannot achieve a sufficient degree of performance, especially in object-oriented languages.

3.1.1 *COOLs in this Category.* Only Mentat and Oz use data dependence analysis to determine whether a method call can be executed concurrently.[3] Some COOLs use data dependence analysis to coordinate access to return values of procedures. This approach, called **wait by necessity**, is further discussed in section 3.2.2.

[3]Mentat has other mechanisms to initiate concurrency, hence the automatic parallelization is not apparent in the appendix.

## 3.2 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.

3.2.1 *Basic Fork and Join.* The **fork** statement is the oldest construct to initiate parallelism at the language level [Conway 1963; Dennis and Van Horn 1966]. 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.

*Discussion.* 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 program is speckled with **fork** and **join** statements. There can be several **join** statements that refer to a single **fork** but only one may be used at run-time. Thus, it is difficult to understand what methods could be executed concurrently at any point of a given program and which side effects might occur. Since the relative execution speeds of different activities are unknown, race-conditions can occur.

3.2.2 *Asynchronous Call and Future.* This and the following section discuss variations of **fork** and **join** that are often used in COOLs. These variations inherit both the advantages and disadvantages of the basic form although some disadvantages are avoided by further reducing the parallel performance.

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**. If the called method has a return parameter, the caller depends on the availability of the return value. One option for dealing with this dependence is an automatic approach, called **wait by necessity** in Eiffel//: The compiler analyzes a given program to determine when the return value really is needed. By automatic insertion of an implicit **join**, the compiler makes sure that the caller only proceeds when the result is available. Because of the limits of data dependence analysis this approach is restricted to obvious cases. In complex situations the compiler falls back to a synchronous call to be defensive and to ensure the intended semantics. Moreover, this approach does not blend smoothly with languages that have a clean exception model that requires explicit catch-clauses.

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 uninitialized future is blocked until another activity writes to the future.

There are several variants of **futures**. The basic futures can only hold a single value. Some COOLs extend this by defining futures as general communication buffers that implement for instance an unbounded queue of result values. An extension in another direction are first-class futures that themselves can be passed as parameters without forcing their evaluation. This is useful to implement call-forwarding: If a particular method cannot provide the return value which it is supposed to write into a future, this method passes on the future to another method that then returns a result to the original caller.

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

Wait-by-necessity reduces parallel performance since it restricts the parallelism to those cases that can be handled by data dependence analysis.

Since calls and futures do not obey the single-entry single-exit paradigm, they tend to be scattered throughout the program. Thus it is hard (a) to understand the set of potentially concurrently executing activities for each point of the source code and (b) to anticipate harmful interferences. Unless the mechanisms are used with care and documentation, programs are difficult to maintain.

3.2.3 *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. The two effects of a classic **return** statement, namely to return a result and to return to the context stack of the caller, are separated.

Post-processing is the natural way of organizing the interplay between activities when methods are invoked by asynchronous message passing (in contrast to procedure calls). If one activity sends an explicit message to an object to invoke one of its methods, the only way to return a result is by sending a message back to the first object. In this case, there is no reason why the second object should reply with the last statement of the method; an earlier reply message results in post-processing.

*Discussion.* Post-processing restricts parallel performance due to limited fanout. Encapsulation can be achieved by client-side coordination provided that the post-processing part of a method is re-entrant and guarantees not to interfere with other activities by only working on private state variables. Since parallelism is restricted to the code lines after the early **return**, it is easier to understand which code could potentially be executed in parallel. This restriction eases debugging.

By means of post-processing existing sequential libraries can gradually be reworked towards a parallel implementation. Such an approach is infeasible for the other two mechanisms where existing sequential, but not re-entrant libraries are hard to use in parallel contexts.

#### 3.2.4 *COOLs in this Category.*

ABCL/x (async. call, 1st class future, post-processing)
Acore (async. call, post-processing)
ACT++ (async. call, 1st class future: [first, last, queue], post-processing)
Act1 (async call, 1st class future [once, queue], post-processing)
Actalk (async. call, post-processing)
ActorSpace (async. call, post-processing)
Actra (post-processing)
A-NETL (async. call, future, post)
Amber (thread object)
ASK (async. call, post-processing)
A'UM (async. call)
Cantor (async. call, post-processing)
CEiffel (async. call, wait by necessity)
CHARM++ (async. call)
CLIX (async. call, post-processing)
Compos. C++ (`spawn` command)
Conc. Aggregates (async. call, post-processing)
Conc. Smalltalk (async. call, 1st class future, post-processing)
cooC (async. call, wait by necessity)
COOL/Chorus (fork)
COOL/NTT (async. call)
COOL/Stanford (async. call)
Coral (async. call)
Correlate (async. call)
CST (async. call, future)
Demeter (thread object)
Distr. C++ (thread object)
Distr. Eiffel (async. call, 1st class future)
Distr. Smalltalk - Object (fork)
Distr. Smalltalk - Process (fork)
DOWL (thread object)
DROL (async. call, post-processing)
Ellie (async. call, 1st class future)
ES-Kit (async. call, manual futures)
ESP (async. call, 1st class futures)
Fragmented Objects, FOG/C++ (async. call, potential for manual futures)
HAL (async. call, post-processing)
Harmony (thread object)
Heraklit (async. call)
HoME (fork)
Hybrid (async. call)
Karos (async. call)
LO (async. call)
Mediators (Life routine, see section 3.5, `spawn` method execution)
MeldC (async. call)
Mentat (async. call, post-processing)
Meyer's Proposal (async. call)
MPC++ (async. call, 1st class future)
Multiprocessor-Smalltalk (fork)
Obliq (fork, join + return value)
Orca (fork)
Parallel Computing Action (async. call, 1st class future)
Parallel Object-oriented Fortran (async. call)
PO (async. call, future)
POOL (post-processing)
Presto (thread object)
Procol (async. call)
pSather (async. call, 1st class future [queue])
PVM++ (thread object)
Python (thread/fork, library)
QPC++ (async. call, wait by necessity, post-processing)
Rosette (async. call, post-processing)
SAM (async. call, post-processing)
Scoop (thread object)
Smalltalk (fork)
SR (async. call, post-processing)
Tool (async. call)
Trellis/Owl (thread object)
Ubik (async. call, post-processing)
UC++ (async. call)

Futures cannot only be used with asynchronous method calls. In the following COOLs they are used as general means for communication and synchronization between parallel activities.

Conc.Class Eiffel
Comp. C++
Distr. C++

Presto

### 3.3 Cobegin

This section covers constructs that may start many concurrent activities at once. These activities are not bound to objects but can operate on the data structures in the same way as the activity that executed the construct.

3.3.1 *Cobegin.* The **cobegin** statement [Dijkstra 1968a] 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 of

**cobegin** $\mathsf{StmtList}_1$ | ... | $\mathsf{StmtList}_n$ **end**

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.

*Discussion.* The construct may break modularity unless encapsulation is preserved by callee-side coordination. **Cobegin** is easy to understand because it restricts the scope of parallel activity to a textual portion of the code by means of an enforced synchronization.

3.3.2 *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, the above mentioned synchronization takes place.

A **cobegin** can be equivalently expressed by means of **par** and **fork**:

**par**
  **fork** ($\mathsf{StmtList}_1$);
  ...
  **fork** ($\mathsf{StmtList}_n$);
**end**

3.3.3 *Activity Set.* The programmer can explicitly add activities to an activity set and then wait for the completion of all those activities.

*Discussion.* Although the effect is similar to the **par** statement, it does no longer provide the ease of understanding of potential concurrency. Whereas the **par** statements narrows the concurrent activities to a couple of program lines, the activity set can be modified anywhere in its scope.

3.3.4 *COOLs in this Category.*

ABCL/x (cobegin)
COOL/Stanford (`waitfor` statement)
Comp. C++ (cobegin)
Conc. Aggregates (cobegin)
DOWL (Activity Set)
Guide (cobegin)
LO (combination join)
Micro C++ (Block = thread boundary)
Proof (cobegin)
pSather (`par` statement, Activity Set)
Rosette (cobegin)
Scheduling Predicates (cobegin)
SOS (cobegin)
Spar (`each` cobegin)
SR (`co`-statement)
Trellis/Owl (Activity Set)

### 3.4 Forall, Aggregate, and Equivalents

This section covers constructs that may start many concurrent activities at once. 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.

3.4.1 *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** statement is intended for a fine granularity and a high degree of parallelism. Although the **forall** statement and its siblings can be understood as being derived from the **cobegin** statement, the new statements bridge the thread-centered understanding of parallelism with the notion of **data-parallel programming**. The data-parallel programmer no longer focuses on threads and on the statements executed by them but thinks in terms of data elements to which operations are applied in parallel.

*Discussion.* The **forall** may break modularity unless encapsulation is preserved by callee-side coordination or by strictly confining concurrency to individual data elements (one per instance of the **forall**), i.e, by avoiding any data dependence within the **forall**. Whereas **cobegin** helps in determining which activities could be executing concurrently, the **forall** statements further reduce complexity by restricting what the activities can do to a single statement list.

3.4.2 *Aggregate.* 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, here the data structuring concept of the aggregate is used.

*Discussion.* As aggregates can be transformed into an equivalent **forall** implementation, the same disadvantages and advantages apply.

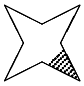
3.4.3 *Multicast and Cluster Aggregates.* Languages that are based on explicit message passing sometimes define aggregates implicitly by defining lists of recipients or by linking several recipients to a single communication channel. This is called **multicast** message passing.

Another form of aggregates is frequently chosen by COOLs that target distributed hardware, e.g., a network of workstations. Here **cluster aggregates** create a single object of a given class on each node of the parallel machine. In contrast to replication, the objects are distinct and have different state. When a member function of a cluster aggregate is called, one instance of the member function executes on each node of the cluster.

3.4.4 *COOLs in this Category.*

ActorSpace (aggregate)
A-NETL (multicast)
Arche (aggregate)
Blaze-2 (forall)
Braid (data-parallel)
C** (data-parallel)
CHARM++ (cluster aggregate)
Comp. C++ (forall)
Conc. Aggregates (aggregate)
dpSather (data-parallel)
EPEE (cluster aggregate)
Fragmented Objects, FOG/C++ (multicast)
HPJava (`over` forall)
Modula-3* (forall)
NAM (data-parallel)
parallel C++ (data-parallel)
Procol (multicast via type)
QPC++ (aggregate: processor set)
Spar (foreach)
SR (array of process: strip, `co`-stmt: quantifier)
Titanium (foreach)

### 3.5 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.

3.5.1 *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.

*Discussion.* The construct restricts parallel performance due to limited fan-out and may break modularity unless encapsulation is preserved by callee-side coordination.

3.5.2 *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 autonomous routines are started automatically upon object creation, other languages require an explicit start of that method.

COOLs that are library-based add-ons to object-oriented languages often offer autonomous routines in a special library class that has the added activity. Other COOLs allow the programmer to explicitly label one or several methods of the class implementation to be autonomous.

*Discussion.* The construct may break modularity unless encapsulation is preserved by callee-side coordination. Additional parallelism requires serial object creation. Autonomous routines are often affected by the library problem.

3.5.3 *Life Routine.* An autonomous routine is called life routine if it is started automatically upon object creation and if it handles incoming method calls or messages that are sent to the object.[4] Life routines have explicit **receive** statements within an endless loop or are equipped with an interrupt mechanism. If the life routine terminates, the object can no longer be used.

*Discussion.* Life routines preserve encapsulation by object-confined concurrency as long as instance variables and functionality code are private. Coordination is isolated in the life routine, resulting in cleaner inheritance. However, since there is one endless loop or one interrupt handler, coordination code is in general not separable. Hence, some inheritance anomalies still occur; exceptions are discussed in section 4. Additional parallelism can only be spawned with serial object creation, i.e., with restricted parallel performance.

3.5.4 *COOLs in this Category.*

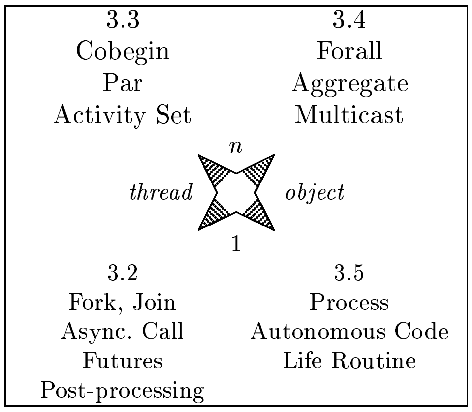
Arche (life routine, self start)
Atom (autonomous routine)
Beta (autonomous routine, separate start)
C++// (life routine, self start)
CEiffel (autonomous routine, self start)
COB (life routine, self start)
Conc. Class Eiffel (life routine, separate start)
Correlate (autonomous routine, self start)
Distr. Eiffel (process, dynamic)
DoPVM (process, static)
Dragoon (autonomous routine, self start)
Eiffel// (life routine, self start)
Emerald (autonomous routine, self start)
Guide (process, static)
IceT (process)
Java (autonomous routine, separate start)
Java//, ProActive PDC (life routine, self start)
Mediators (life routine, self start)
Mentat (life routine, self start)
Micro C++ (autonomous and life routine, self start)
Moose (autonomous routine, self start)

[4]Some authors prefer the word "active" instead of "life". We do not take on this phrasing since the term "active objects" is heavily overloaded in the literature.

Panda (autonomous routine, self start)
POOL (life routine, self start)
Proof (autonomous routine, self start)
QPC++ (life routine, self start)
SR (autonomous and life routine, self start)
Titanium (process)

### 3.6 Bird's-eye view

The constructs mentioned in the upper half of the following diagram have a high fan-out and are thus advantageous for achieving parallel performance. The later a construct has been presented above, the easier it is for the programmer to understand what concurrent activities might exist and what code these activities might be executing. All mechanisms for initiation of concurrency need callee-side concurrency coordination to avoid breaking encapsulation.




## 4. COORDINATING CONCURRENCY

This section surveys coordination constructs found in COOLs. The main distinction is made with respect to the goal of callee-side coordination, as discussed in section 2.2. Activity-centered coordination mechanisms are discussed in section 4.1; boundary coordination mechanisms and subgroups are presented in section 4.2.

*Structure of the Discussions.* Below, individual language constructs are studied with respect to the goals given in section 2. This is the usual structure:

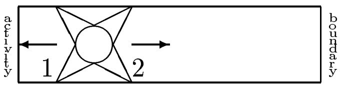
callee-side: We analyze whether the coordination is enforced at the side of the callee. Activity-centered coordination mechanisms usually do not fulfill this goal, whereas boundary coordination mechanisms do.

expressive: We check whether the mechanisms allow restricted intra-object concurrency. Moreover, it is discussed whether state proceed-criteria and history proceed-criteria can be expressed easily.

isolated: For boundary coordination, we discuss whether the proceed-criteria are isolated from functionality code because otherwise inheritance anomalies must be expected. For activity-centered coordination mechanisms it depends on the programming style whether the coordination code is isolated (typically it is not).

separable: Similarly, we check whether proceed-criteria can be expressed and inherited separately since that would help avoid inheritance anomalies.

### 4.1 Activity-Centered Coordination




In the pictographs, languages with activity-centered coordination are represented by a circled star on the left of the slide-bar, typically at position 1. In general, mechanisms in this group do not fulfill the goal of callee-side coordination though some allow fulfilling it manually.

4.1.1 *Synchronization by Termination.* Most of the mechanisms for initiating parallelism (see section 3) 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. Consider the example:

```
forall i:[range] do a[i] := i end;
forall i:[range] do b[i] := a[i-1] end
```

At first, all concurrent activities initialize "their" element of a. Afterwards, they set "their" element of b to the value stored in the "neighboring" element of a. Concurrency is coordinated by termination: Since after each assignment the activities terminate, those from the second **forall** cannot interfere with those from the first **forall**.

*Discussion.*

| | |
|---|---|
| callee-side: | no |
| expressive: | intra-object concurrency |
| isolated: | n/a |
| separable: | n/a |

4.1.2 *Semaphore, Mutex, Lock.* The semaphore is another basic concept of organizing concurrent access to shared data [Dijkstra 1968a; Dijkstra 1968b]. 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 P or wait operation blocks until the variable is greater than zero in which case the variable is decreased atomically. The other operation, called V or signal, increases the variable atomically. When the value of the variable is greater than 1, 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.

*Discussion.*

| | |
|---|---|
| callee-side: | no, but can be achieved manually |
| expressive: | intra-object concurrency |
| isolated: | n/a |
| separable: | n/a |

4.1.3 *Conditional Critical Region.* Conditional critical regions provide syntactic support for conditional coordination of parallelism [Hansen 1972; Hansen 1973a; Hoare 1972]. Whereas in critical regions defined by semaphores arbitrary code can be executed and hence accesses to arbitrary sets of data elements can be coordinated, conditional critical regions make the purpose of coordinating accesses more transparent. The idea is to collect variables in so-called resources. Every variable $v_i$ can belong to at most one resource.

**resource** r **is** $v_1, v_2, \ldots, v_m$;

Afterwards a critical region is declared:

**region** r **when** Condition
**then** StmtList **end**

Only when the Condition holds and no other activity is in a region of the same resource, an activity can execute StmtList. Otherwise the activity blocks until the condition holds and the resource is free.

*Discussion.* Only one activity can enter a region at a time. But with small regions some intra-object concurrency is possible. Conditional critical regions in general do not fulfill the goal of callee-side coordination. However, it can be achieved manually, if the scope of the resource and the StmtList are restricted to only class attributes.

With the **when** clause of the **region** statement it is much easier to implement state proceed-criteria than with semaphores. In comparison to semaphores, conditional critical regions solve a few of the problems introduced by the generality of semaphores: It is impossible to forget to close a critical region and it is easier to implement conditional coordination since the conditions are explicit. Other hard problems remain. It is still easy to run into deadlocks and the code that works on variables from a particular resource is still spread over the class.

| | |
|---|---|
| callee-side: | no, but can be achieved manually |
| expressive: | no intra-*region* concurrency, |

| | |
|---|---|
| | state proceed-criteria |
| isolated: | n/a |
| separable: | n/a |

4.1.4 *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.

**receive** m [**from** s]

*Discussion.* Because the callers must know the exact calling protocol, i.e., implementation details of the called object, piggy-backed synchronization does not fulfill the goal of callee-side coordination. Since piggy-backed synchronization mechanisms can only accept a single message at a time, these constructs cannot be used to coordinate intra-object concurrency. Only state proceed-criteria can be implemented.

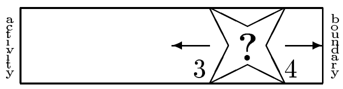
| | |
|---|---|
| callee-side: | no |
| expressive: | no intra-object concurrency, state proceed-criteria |
| isolated: | n/a |
| separable: | n/a |

4.1.5 *COOLs in this Category.* COOLs with asynchronous method calls, messages, and futures (section 3.2) offer some form of piggy-backed synchronization. We do not repeat these languages in the list below.

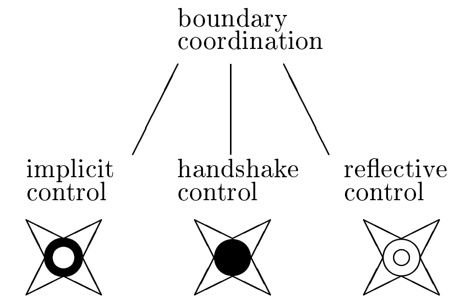
Amber (lock, barrier)
Atom (enable set, instead of "become" like guarding conditions)
Beta (semaphore)
Blaze-2 (termination, lock)
Braid (termination)
C** (termination)
Comp. C++ (coordination future)
Conc. Aggregate (reader/writer lock)
Conc. Smalltalk (semaphore)
cooC (semaphore)
COOL/Chorus (semaphore)
CST (semaphore)
Distr. C++ (coordination future)
Distr. Eiffel (semaphore, lock)
Distr. Smalltalks (semaphore)
DoPVM (lock)
DOWL (lock)
dpSather (termination)
EPEE (termination)
ES-Kit (lock)
Harmony (semaphore)
HoME (semaphore)
HPJava (termination, piggy backed sync.)
IceT (piggy backed sync.)
Java (mutex)
Karos (termination)
LO (termination)
MeldC (mutex, semaphore)
Modula-3* (termination)
MPC++ (mutex)
Multiprocessor-Smalltalk (semaphore)
NAM (termination)
Obliq (mutex, lock)
Panda (semaphore)
parallel C++ (termination)
PO (semaphore)
Presto (lock, mutex, coordination future)
Proof (lock)
pSather (lock)
PVM++ (lock, semaphore)
Python (semaphore, mutex, lock)
QPC++ (semaphore)
Scoop (piggy backed sync.)
Smalltalk (semaphore)
Spar (termination, mutex)
SR (semaphore)
Titanium (termination, piggy backed sync.)
Trellis/Owl (lock)

Note that several COOLs in this category, e.g. pSather, intentionally accept the problems of activity-centered coordination in favor of performance since it is well-known how to implement activity-centered coordination mechanisms efficiently.

### 4.2 Boundary Coordination




In the graphical notation, languages with boundary coordination (that fulfill the goal of callee-side coordination) are represented by a star on the right of the slide-bar. Depending on the type of coordination, the star is refined. 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.




*Implicit Control.* The language prohibits intra-object concurrency and defines for all classes which of several concurrent method invocations to execute next. Since the run-time system implements this rule, no explicit coordination code is required. See section 4.2.1.

*Handshake control.* The dynamic interface of the class can be modified by explicit coordination code in the class implementation. This code can express which method invocation is to be delayed by the run-time system and when to allow intra-object concurrency. Hence, class implementation and run-time system work hand in hand.[5] The language constructs proposed in COOLs heavily differ with respect to isolation, separability, and expressiveness of coordination code. See sections 4.2.3 to 4.2.5.

*Reflective control.* The dynamic interface of the class can again be modified and some intra-object concurrency can be allowed. But in contrast to handshake control the programmer can provide the coordination code in an (associated) meta-class that is consulted by the run-time system whenever coordination is required. (Some authors claim that the meta-class almost belongs to the run-time system.) Mechanisms in this group offer isolated coordination code; they vary with respect to separability and expressiveness. See section 4.2.6.

[5]In COOLs based on message passing instead of method invocations, the input queues are handled by the class. Since the other end of the queues is handled by the run-time system, we treat the coordination mechanisms as handshake control as well.

4.2.1 *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. An object can neither influence which method call to execute if several arrive concurrently nor in what order.

*Discussion.* Mechanisms in this group fulfill the goal of callee-side coordination but lack intra-object concurrency. The key advantage of this restriction is that the definition of local correctness can be used without alterations. It is thus not more difficult to implement correct classes than it is in sequential object-oriented languages.

4.2.1.1 *Monitor.* Implicit control is an instance of the monitor concept [Dijkstra 1968a; Hansen 1973b; Hoare 1974]. 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. In some COOLs all objects are monitors by definition; in other COOLs the programmer can identify monitor classes.

*Discussion.* Monitors forbid intra-object concurrency and are not made to express proceed-criteria. Since there is no explicit coordination code, the goals of isolated and separable coordination code are trivially met.

callee-side: yes
expressive: no intra-object concurrency
isolated: yes
separable: yes

Monitors inherit another disadvantage from the underlying semaphores: careless usage can still result in deadlocks, cf. **nested monitor call problem** [Haddon 1977; Lister 1977; Wettstein 1978].

To increase expressiveness, extensions of the monitor concept have been proposed. But they no longer have isolated and separable coordination code.

4.2.1.2 *Condition Variables.* In this monitor extension [Hoare 1974], an activity that has entered a monitor can block inside of the monitor 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.

*Discussion.* While state proceed-criteria can now be expressed, the goals of isolated and separable coordination code no longer hold since coordination code is mixed into the methods that implement functionality.

callee-side: yes, if condition variables are private attributes of the class
expressive: no intra-object concurrency, state proceed-criteria
isolated: no
separable: no

4.2.1.3 *Conditional Wait.* This variant of condition variables has been introduced by [Kessels 1977] to improve the conditional synchronization in monitors. Kessels proposed to isolate the conditions syntactically instead of spreading **wait** and **signal** over the class.

**condition** identifier : cond-expr;

With this declaration of a condition, an activity can wait simply by calling **wait**(identifier) and relying on the run-time system to signal the continuation when cond-expr becomes true. 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.

*Discussion.* Conditional wait fulfills the goal of callee-side coordination code if the condition is a private part of the class. Moreover, the condition itself must be computed solely based on the internal state of the object since otherwise a calling activity could influence the coordination from the outside of the object. Since **wait** commands can exist everywhere in the code, some inheritance anomalies can occur.

callee-side: yes, in certain conditions.
expressive: no intra-object concurrency, state proceed-criteria
isolated: no
separable: no

If the conditions use many instance variables, the conditions must be checked frequently at run-time, i.e., whenever one of the instance variables is changed. This can degrade performance.

4.2.1.4 *COOLs in this Category.*

Amber (monitor)
A-NETL (monitor)
A'UM (monitor)
CHARM++ (monitor)
Conc. Smalltalk (monitor)
COOL/NTT (monitor)
COOL/Stanford (condition variable)
CST-MIT (monitor)
Emerald (monitor)
ESP (monitor)
Fleng++ (monitor)
Fragmented Objects, FOG/C++ (monitor)
Heraklit (defer[6], conditional wait)

[6]A special form of condition variable is offered if a method can **defer** its own execution. In terms of condition variables, deferring is equivalent to

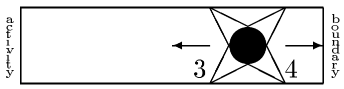
Mentat (monitor, sequential and persistent objects)
Micro C++ (monitor)
Obliq (condition variable)
Orca (monitor)
Oz, Perdio (monitor)
Panda (monitor)
Presto (condition variable)
SAM (monitor)
Tool (monitor)
UC++ (monitor)

4.2.2 *Running Example.* A bounded buffer is the illustrative example used throughout the rest of section 4. The buffer is implemented with a fixed-size array. Several clients can put objects into the buffer or get objects from the buffer in a FIFO order. Invocations of put or get must be delayed if the buffer is full or empty, respectively. Assume that the buffer implementation provides two private methods isEmpty and isFull that return the appropriate boolean value. For brevity, we ignore the implementation details of the buffer but only show the coordination code.

```
class BBUFFER is
public interface:
  put(t:OBJECT);
  OBJECT:get();
implementation: -- see below
end BBUFFER;
```

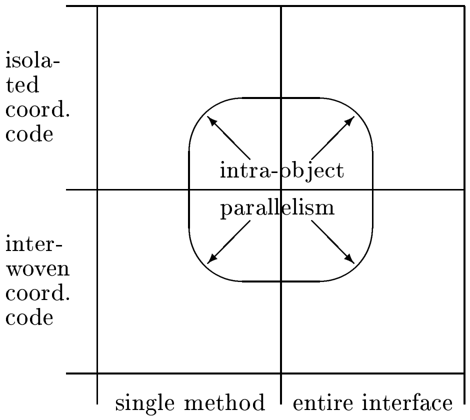
The following exercise is recommended to the reader: for each of the situations below, try to add a method get2 that returns two elements from the buffer. Originally, the bounded buffer has three different states. A fourth state is needed for an element count of one. It is instructive to experiment with the resulting inheritance anomalies.

4.2.3 *Handshake Control.*




Boundary coordination with handshake control (see above pictograph) divides the responsibility for coordination between the object's implementation and the run-time system (or the handler of message queues). 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. The run-time system reacts according to this specification.

Handshake control mechanisms fulfill the goal of callee-side coordination; they differ with respect to the other goals.




We further refine the group of handshake control mechanisms along the three dimensions of the above diagram.

—**Interwoven Coordination Code.** COOLs that implement the coordination in the body of public methods fall into the lower half of the diagram. Mechanisms in this group do not fulfill the goal of isolated coordination code; they are discussed in section 4.2.4. The star is at position 3 (or 1 if the COOL offers additional activity-centered coordination mechanisms.)

—**Isolated Coordination Code.** COOL mechanisms in the upper half (see section 4.2.5) have isolated coordination code. The star is at position 4 (or 2).

---

waiting at a condition variable with a guaranteed and immediate signaling. The execution of the method is interrupted and returned to the run-time system for later re-scheduling.

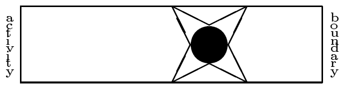
—**Intra-Object-Concurrency.** A second binary dimension distinguishes COOLs that are based on the one-activity-at-a-time principle (outer part of the diagram) from COOLs that allow concurrent execution of methods (inside of the oval). This distinction is not visible in the pictographs.

—**Single Method.** On the left hand side we discuss COOL coordination mechanisms that affect the object's dynamic interface with respect to a single method, i.e., before a method is executed, proceed-criteria are checked that solely belong to this method.

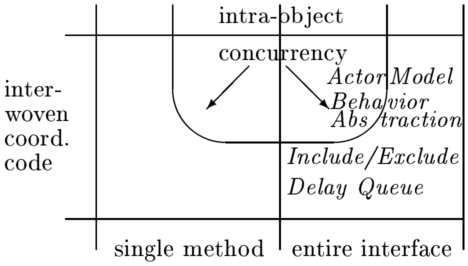
—**Entire Interface.** On the right hand side are COOL coordination mechanisms that affect the entire dynamic interface. The code that enables and disables methods is in no direct and close relation to the affected method. The distinction between these two categories is also not shown in the pictographs.

The subsequent discussion of handshake control mechanisms follows the isolated/interwoven dimension because this dimension has a strong impact on inheritance anomalies (see section 2.3).

4.2.4 *Interwoven Handshake Control*




The following diagram names the mechanisms that are discussed in this section. They fulfill the goal of callee-side coordination, but fall short to meet the goal of isolated coordination code and thus cause inheritance anomalies. Since the coordination code is not isolated, the goal of separable coordination code cannot be met. All four mechanisms are based on the one-activity-at-a-time principle; the two whose name reaches into the oval of intra-object concurrency allow post-processing (or can be extended to allow it). All mechanisms affect the entire dynamic object interface. The proceed-criteria of a method are not tightly connected to a particular method. Instead proceed-criteria of one method are computed in other methods.




4.2.4.1 *Delay Queues.* They can be linked to methods that are to be hidden from an object's interface. Delay queue objects have two methods in their public interface, namely `open` and `close`. If the delay queue is open, a call of the method can proceed, otherwise it is delayed.

Arbitrary proceed-criteria can be implemented by conditional opening and closing of delay queues. Multiple delay queues can be used and several methods can be linked to the same delay queue.

*Example.* To implement the bounded buffer, we need two delay queues (`putQ` and `getQ`). The method `put` or `get` are linked to `putQ` or `getQ`, respectively. At the end of each method, the status of the delay queues is set according to the element count of the buffer.

An additional `counter` method would not need to touch the delay queues.

```
class BBUFFER is
public interface: -- see above
implementation:
  private putQ, getQ : DELAYQUEUE;
  put(t:OBJECT) link putQ is ...
    getQ.open();
    if isFull() then putQ.close();
  end;
  OBJECT:get() link getQ is ...
    putQ.open();
```

```
    if isEmpty() then getQ.close();
  end;
end BBUFFER;
```

*Discussion.* Delay conditions fulfill the goal of callee-side coordination, provided that delay queues are private attributes of the class that is accessed concurrently. Delay queues need the one-activity-at-a-time principle. Otherwise in the example, concurrent execution of put and get could result in illegal situations, e.g., an open getQ with an empty buffer.

| | |
|---|---|
| callee-side: | yes, if delay queues are private |
| expressive: | no intra-object concurrency, state proceed-criteria |
| isolated: | no |
| separable: | no |

4.2.4.2 *Include/Exclude.* **Include** or **exclude** commands add or remove methods explicitly from an object's dynamic interface. Include/exclude implement proceed-criteria: When a method is called that is currently excluded from the dynamic interface, the call is delayed until the method is included again.

Programs with include/exclude tend to be more verbose than with delay queue mechanisms since there is no way to combine methods with similar coordination constraints into one statement. Instead all these methods must be included and excluded individually.

*Example.* Initially, only put can be called. In the public interface section of the code this is indicated by the additional key word **initial**. After the first element has been put into the buffer, both put and get can be called. Similarly, put is excluded from the active interface if the buffer is full.

```
class BBUFFER is
public interface:
  initial put(t:OBJECT);
  OBJECT:get();
implementation:
  put(t:OBJECT) is ...
    include {get};
    if isFull() then exclude {put};
  end
  OBJECT:get() is ...
    include {put};
    if isEmpty() then exclude {get};
  end
end BBUFFER;
```

Although coordination code is somewhat separated from the functionality code the **include** and **exclude** commands are still mixed into the functionality code.

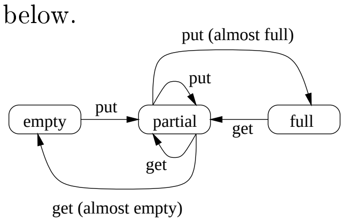
An additional counter method would leave the active interface unchanged.

*Discussion.* Include/exclude is based on the one-activity-at-a-time principle; this cannot be waived.

| | |
|---|---|
| callee-side: | yes |
| expressive: | no intra-object concurrency, state proceed-criteria |
| isolated: | no |
| separable: | no |

4.2.4.3 *Behavior Abstractions.* While delay queues and include/exclude change the object interface directly, behavior abstractions introduce the additional concept of "state". The programmer is required to conceive the object behavior in form of a finite state automaton. In each state, one of several methods is callable although only one method can be invoked at a time. The **become** statement is used to switch to another state and to return from a method.

*Example.* The bounded buffer has three states, namely `empty`, `partial`, and `full`. In the diagram below, the states are represented by ovals. The arrows mark acceptable methods that are declared in the **behavior** section of the code below.



```
class BBUFFER is
public interface: -- see above
behavior:
  empty   = {put}
  partial = {put, get}
  full    = {get}
implementation:
  put(t:OBJECT) is ...
    if isFull() become full
    else become partial;
  end
  OBJECT:get() is ...
    if isEmpty() become empty
    else become partial;
  end
end BBUFFER;
```

*Discussion.* Whereas delay queue operations, includes, and excludes are spread across the class and make it difficult to reason about which methods are hidden at any given time, behavior abstractions make the set of available methods visible at method termination time. They need an **if**-cascade at the end of the methods for deciding to which state to switch. That causes inheritance anomalies if a state is partitioned.

| | |
|---|---|
| callee-side: | yes |
| expressive: | no intra-object concurrency, post-processing feasible, state + history proceed-criteria |
| isolated: | no |
| separable: | no |

Behavior abstractions rely on the one-activity-at-a-time principle for the same reasons discussed for delay queues and include/exclude. In contrast to those mechanisms however, it is easier to weaken the restriction here. Since there is a single point in the implementation of a method that determines the new state, a concurrent method could be started immediately afterwards, even if the first method has not yet terminated. This is an application of post-processing. The increased parallelism sacrifices the goal of callee-side coordination unless the post-processing part guarantees that the class invariant is fulfilled at any time during its execution.

Although coordination code is more isolated from the functionality code, there is still coordination code mixed into the functionality code, namely the **become** statements and the static encoding of the following state.

4.2.4.4 *Actor Model.* The pure Actor model [Agha 1986] combines the concept of objects with the concept of dynamic interfaces. When the dynamic interface of an actor, i.e. an object, is changed, this actor becomes a new actor with a different behavior, albeit with the same values of the instance variables. Instead of dynamically changing whether a method can be called or not, in the Actor model the code that implements the actor behavior is switched dynamically.

*Example.* In languages based on the Actor model, each of the three states of the buffer is implemented separately. In some Actor languages the **become** statement can be omitted if the current state is not changed. To indicate that, the above code has lines in brackets.

```
Actor EmptyBB is
public interface:
  put(t:OBJECT);
implementation:
  put(t:OBJECT) is ...
    become PartialBB(state);
  end
end EmptyBB; -------------------------
Actor PartialBB is
public interface:
  put(t:OBJECT);
  OBJECT:get();
implementation:
  put(t:OBJECT) is ...
    if isFull() become FullBB(state)
    [else become PartialBB(state)];
  end
  OBJECT:get() is ...
    if isEmpty() become EmptyBB(state)
    [else become PartialBB(state)];
```

```
  end
end PartialBB; -----------------------
Actor FullBB is
public interface:
  OBJECT:get();
implementation:
  OBJECT:get() is ...
    become PartialBB(state);
  end
end FullBB;
```

*Discussion.* The Actor model uses post-processing: After a **become** statement, a new method can be started while the first method can continue.

| | |
|---|---|
| callee-side: | yes |
| expressive: | intra-object concurrency by means of post-processing, state + history proceed-criteria |
| isolated: | no |
| separable: | no |

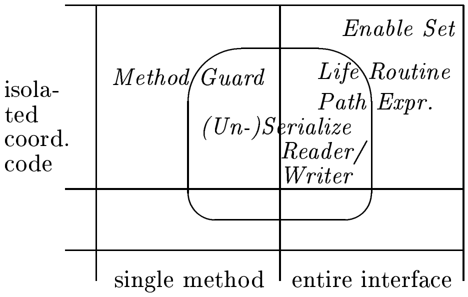
Some extended Actor models have the notion of **unserialized methods** that are re-entrant, i.e., that can be executed by several callers concurrently. An unserialized method does not change the behavior of the actor, i.e., the **become** statement sets the current state again. In the example, the additional method counter could be an unserialized method. We further discuss the issue of serialized/unserialized methods in more detail in section 4.2.5.

4.2.4.5 *COOLs in this Category.*

ABCL/1 (Actor)
ABCL/f (Actor)
Acore (Actor)
ACT1 (Actor)
Actalk (Actor)
ActorSpace (Actor)
Actra (Actor)
ASK (Actor)
Cantor (Actor)
Distr. C++ (delay queue)
Ellie (include/exclude)
Hybrid (delay queue)
Parallel Object-Oriented Fortran (include/exclude)
Ubik (Actor)

### 4.2.5 *Isolated Handshake Control*


The following diagram names the mechanisms discussed in this section that fulfill the goal of callee-side coordination code and are closer to fulfilling the goal of isolated coordination code than previous mechanisms. Some mechanisms meet the goal of separable coordination code; some allow intra-object concurrency. Note that three mechanisms are located both inside and outside of the oval since they – although originally designed for the one-activity-at-a-time principle – have straightforward extensions to handle intra-object concurrency.




4.2.5.1 *Method Guard.* With method guards, proceed-criteria can be expressed similar to pre-conditions. Before a guarded method is executed its condition is evaluated. If it holds, the method is invoked, otherwise the call is delayed. The condition can be an expression over all instance variables of the object.

*Example.* There is a special **guard** section in the class that specifies proceed-criteria for every method. The methods put or get can only be executed when the buffer is not full or not empty, respectively.

```
class BBUFFER is
public interface: -- see above
guards:
  put: !isFull()
  get: !isEmpty()
implementation:
  put(t:OBJECT) is ... end
  OBJECT:get()  is ... end
end BBUFFER;
```

*Discussion.* Although the coordination code seems to be isolated from the code that implements functionality, there are still interdependencies and thus inheritance anomalies. Since the guards use instance variables to check the element counter, the coordination code is connected to the functionality code that uses the same instance variables. If the instance variables are changed, the outcome of the conditions might change as well. However, isolation of coordination code is much better than in the mechanisms discussed earlier since methods and guards can be inherited separately. To further reduce inheritance anomalies, [Frølund 1996] has proposed to use negative guards that express when a method cannot be called. Subclasses do not overwrite the guards but instead combine the negative guards: all guards along the inheritance chain must evaluate to false before a method can be called.

| | |
|---|---|
| callee-side: | yes |
| expressive: | rarely intra-object concurrency, state proceed-criteria |
| isolated: | almost yes |
| separable: | yes |

Method guards are often based on the one-activity-at-a-time principle. Sometimes it is expressible that a method can/cannot be executed while (other) methods are executed concurrently. There are two different proposals for it:

*Counters* are predefined functions to be used in the guarding conditions. For example, there might be a counter for the number of currently active method executions (in total or for a specific method). Other counters return the number of pending invocations, of completed method executions, and so on. With these counters, a guard expression can check concurrency conditions.

*Compatibilities* are a feature of CEiffel. In a method guard the programmer can specify the names of methods that are allowed to execute concurrently. When a method is called, the run-time system checks whether a method is executed that is not in the list of concurrently executable methods. If so, the call is delayed; otherwise it proceeds.

Conditions that use many instance variables must be checked frequently at run-time, i.e., whenever one of the instance variables is changed. This can degrade performance.

4.2.5.2 *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.

Enable sets ease this problem: Instead of a **become** statement that requires the name of a state, enable sets are first class citizens of the language. Hence, the programmer can call a method in the **become** statements that returns the new state. The complex analysis is then hidden in this method. When the sets of potential states change, hopefully only this method is affected. [Kafura and Lavender 1996] reason that first class citizenship of enable sets is a requirement for avoiding inheritance anomalies.

*Example.* The **become** statements in the public methods call the private method next that returns an enable set. (It would be better but longer to have separate next methods for put and get to reduce inheritance anomalies.)

```
class BBUFFER is
public interface: -- see above
implementation:
  private EnableSet:next() is
    if isFull()
      return new EnableSet(get);
    if isEmpty()
```

```
      return new EnableSet(put);
    return new EnableSet(put,get);
  end
  put(t:OBJECT) is ...
    become next();
  end
  OBJECT:get() is ...
    become next();
  end
end BBUFFER;
```

*Discussion.* Enable sets are not made for intra-object concurrency, and this restriction cannot be removed. The reason is the same as for delay queues, include/exclude, and behavior abstractions. Compared to behavior abstractions, enable sets almost fulfill the goal of isolated coordination code since determination of the successor state is moved to separate methods. At this new position, the coordination code can be inherited and modified separately, usually without affecting the methods that implement class functionality. Some problems remain if the state-determining method uses class attributes that are used by other methods as well. In this case the same mild forms of inheritance anomaly can be noticed that have been discussed for method guards.

Since it is possible to achieve a one-to-one mapping between (regular) methods and private state-determining methods, the coordination code is separable.

| | |
|---|---|
| callee-side: | yes |
| expressive: | no intra-object concurrency, state + history proceed-criteria |
| isolated: | almost yes |
| separable: | yes |

*Extension: Default Transition.* At the end of each method, the new state must be determined. Hence, an extension of enable sets is to identify a default method that is called automatically for this purpose. This removes the **become** statement from the language. If a functionality method needs a different algorithm for determining the successor behavior, there are syntactic means to link it to an additional transition method. With respect to inheritance, this extension performs slightly better than classic enable sets.

4.2.5.3 *Path Expression.* A growing monitor code causes both sequential bottlenecks and code for conditional synchronization that is spread over the class. A path expression [Campbell and Habermann 1974] is a single collection of all dependences between potentially concurrent operations.

**path** path_list **end**

The path list is essentially a list of method names, enhanced in a regular expression style, i.e., choice (|), repetition ({}), concurrency (*) etc. can be expressed. The path list specifies which operations can be called in what order, and which operations can be executed concurrently.

*Example.* There is no generally applicable path expressions for the running example. The following code assumes a buffer size of 3.

```
class BBUFFERsize3 is
public interface: -- see above
path:
  (put,get|(put,get|(put,get)*)*)*
implementation:
  put(t:OBJECT) is ... end
  OBJECT:get()  is ... end
end BBUFFERsize3;
```

*Discussion.* Whereas sometimes path expressions seem elegant, they do not always allow to encode the intended semantics because they inherit the monitor's deficiency to properly express conditional synchronization [Bloom 1979].

The main inheritance problem of path expressions is their lacking separability. A subclass can either inherit the whole path expression or completely redefine it. There is no way to just alter a part of a path expression.

| | |
|---|---|
| callee-side: | yes |
| expressive: | intra-object concurrency, history proceed-criteria |
| isolated: | yes |
| separable: | no |

4.2.5.4 *Life Routine.* The basics have been discussed in section 3.5.3. Whereas previous mechanisms use a declarative specification, life routines specify coordination procedurally. The constructs discussed below use the message terminology instead of understanding messages as method calls.

*Receive Statement.* This can be used to explicitly wait for the arrival of a message. Often an additional condition or the sender can be specified (**receive** ... **when** or **receive** ... **from**).

*Guarded Commands.* [Dijkstra 1975]: If, in the **if** statement below, one of the boolean conditions (guards) $\mathsf{G}_1$–$\mathsf{G}_n$ holds, the corresponding list of statements is executed; for example a particular message can be received.

**if** $\mathsf{G}_1 \longrightarrow \mathsf{StmtList}_1$
[] $\mathsf{G}_2 \longrightarrow \mathsf{StmtList}_2$
...
[] $\mathsf{G}_n \longrightarrow \mathsf{StmtList}_n$
**end**

If several guards hold, then one of them is selected randomly. Guarded commands are often provided in the syntactic form of **select** statements.

*Example.* If put is called for a buffer that is not full, the **select** statement enters its first branch. Inside this branch the method my_put is started with a **fork** statement. Therefore, the life routine is immediately able to accept the next incoming message. Calls of get are not spawned for concurrent execution; while my_get is processed, further method calls will be delayed.

```
class BBUFFER is
public interface: -- see above
implementation:
  life body is
    loop -- forever
      select
      [] !isFull()
      -> receive "put(t:OBJECT)";
         fork my_put(t);
      [] !isEmpty()
      -> receive "get()";
         my_get();
      end;
    end;
  end;
  my_put(t:OBJECT) is ... end
  OBJECT:my_get()  is ... end
end BBUFFER;
```

*Discussion.* Life routines fulfill the goal of callee-side coordination code. Coordination code is isolated, provided that all coordination code and no functionality code is in the life routine. Intra-object concurrency can be added to life routines: The life routine (a) uses for example a **fork** command to execute a requested method concurrently and (b) checks whether an incompatible method is currently processed. Coordination code is not separable. Below we discuss an extension of life routines for cleaner inheritance.

| | |
|---|---|
| callee-side: | yes, in life routine |
| expressive: | intra-object concurrency, state proceed-criteria |
| isolated: | yes |
| separable: | no |

It is challenging to implement life routines without polling or to evaluate guards as rarely as possible to achieve good performance.

*Extension: Standard Life Routine.* In Eiffel// standard life routines can be inherited from a library of generalized life routines. The programmer provides both the functionality methods and boolean guard functions. To use the inherited life routine, the programmer must initialize a table that stores information about the combination of guard function and method. The prerequisite of inherited life routines is that methods are first class citizens of the language, otherwise functions cannot be table entries.[7] Since there is no longer explicit coordination code, subclasses inherit the intended life behavior implicitly. Often only very few table entries must be modified in the subclass. This is very similar to boundary coordination with implicit control, where the coordination constraints are implicit and thus do not interfere with inheritance.

Since methods are stored in a table, it is difficult to optimize away dynamic method dispatch or to do inlining. This may degrade performance in some COOL implementations.

4.2.5.5 *(Un-)Serialized Method and Object.* Several COOLs that otherwise rely on the one-activity-at-a-time principle allow labeling of methods or even objects as "unserialized". Unserialized methods are purely functional and free of obtrusive side effects and can be executed by several concurrent activities.

The labeling is a step towards the expressibility of intra-object concurrency, but there is no way to express for example the mutual incompatibility of two re-entrant methods.

COOLs with (un-)serialized methods or objects often have an additional activity-centered coordination mechanism. The language list below names that mechanism where appropriate.

*Discussion.* (Un-)serialized methods fulfill most goals, except that they offer very limited expressiveness. Because of this restriction, this construct is often combined with another mechanism; then the other mechanism determines whether and which goals are met.

[7]With these prerequisites, the same approach can used as a programming style.

callee-side: yes
expressive: some intra-object concurrency
isolated: yes
separable: yes

4.2.5.6 *Reader/Writer Protocol.* Methods can be labeled readers (sometimes called observers) or writers (modifiers). The run-time system can then ensure that modifying methods have exclusive access to the object whereas several observing methods can be executed concurrently.

*Discussion.* See section 4.2.5.5 for a discussion of (un-)serialized methods.

callee-side: yes
expressive: some intra-object concurrency
isolated: yes
separable: yes

4.2.5.7 *COOLs in this Category.*

Acore (unserialized method)
ACT++ (enable set, called: behavior set)
ASK (serialized)
Arche (enable set, reader/writer)
Blaze-2 (serialized method, also: lock)
C++// (life routine, 1st class methods)
CEiffel (method compatibility, method guard)
CLIX (method guard)
COB (life routine)
Comp. C++ (serialized, also: coordination future)
Conc. Aggregate (unserialized, also: reader/writer lock)
Conc. Class Eiffel (life routine)
cooC (serialized, also: semaphore)
COOL/Stanford (serialized, also: condition variable)
Correlate (method guard)
Demeter (serialized, method guard)
Distr. Eiffel (method guard, reader/writer)
Distr. Smalltalk – Process (method guard, serialized, also: semaphore)
Dragoon (method guard, counter)
Eiffel// (life routine, 1st class methods)
Guide (method guard, counter)
HAL (method guard)
Java (serialized, also: mutex)
Java//, ProActive PDC (life routine, 1st class methods)
Mediators (life routine, receive, method guard, counter)
Mentat (life routine, receive)
Meyer's Proposal (method guard)
Micro C++ (life routine, receive)
Moose (method guard)
Obliq (serialized, also: mutex, lock)
Orca (method guard)
Parallel Computing Action (method guard)
PO (method guard)
POOL (life routine, receive)

Procol (path expression, method guard)
Proof (method guard)
QPC++ (life routine, receive)
Rosette (serialized, enable set)
Scheduling Predicates (method guard, counter)
SOS (method guard, counter)

Note that some COOLs offer two coordination mechanisms. For example, Procol offers both path expressions and method guards. Both serialized methods and reader/writer can easily be combined with other mechanisms.

4.2.6 *Reflective Control*


COOLs 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.

*Example.* In addition to the unmodified buffer from section 4.2.2, there is a meta-class with two methods in its interface, namely entry and exit.

```
class BBUFFER_SHADDOW is
public interface:
  entry (method-id) is ... end;
  exit  (method-id) is ... end;
end BBUFFER_SHADDOW;
```

Before a buffer b can be used, the run-time system must be informed that its coordination is handled by an object bshadow of the meta-class.

```
SYSTEM::attach(b,bshadow);
```

For an incoming method call for b, the run-time system then first starts the method entry of the shadow object, then invokes the called method of b, and finally calls the method exit of the shadow object. The shadow object can thus implement any form of delay.

*Discussion.* Reflective control mechanisms fulfill the goals of callee-side coordination and isolated coordination code. If the shadow object can be used like a standard life routine and the programmer refrains from implementing functionality in the shadowing class, the coordination code is separable. Otherwise it is not: To change the coordination constraints, the shadowing class must be completely re-programmed.

| | |
|---|---|
| callee-side: | yes |
| expressive: | intra-object concurrency, state proceed-criteria |
| isolated: | yes |
| separable: | no, possibly yes. |

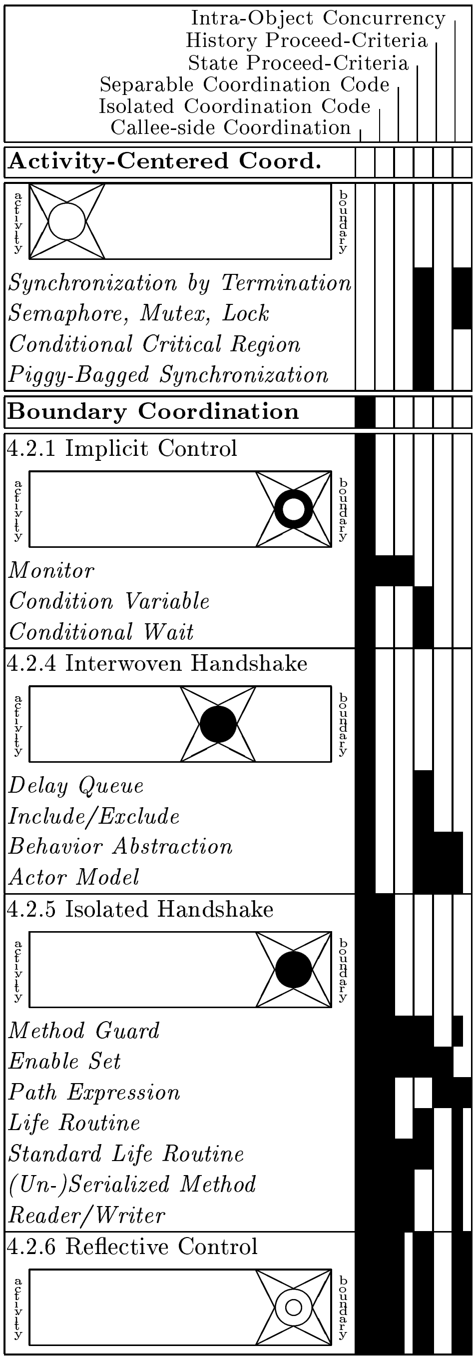
Since each object is shadowed by a second object, storage consumption and object creation removal might noticeably degrade performance.

4.2.6.1 *COOLs in this Category.*

ABCL/R2
ABCL/R3
DROL (protocol object, 1-activity/time)
HAL
MeldC (shadow object, intra-object conc.)

## 4.3 Bird's-eye view

Whereas activity-centered coordination mechanisms do not fulfill the goal of callee-side coordination, mechanisms for boundary control do. None of the mechanisms discussed can fulfill all goals, some of the mechanisms in the lower part of the diagram do considerably better than the monitor. Mechanisms based on interwoven handshake control do not have isolated coordination code, which causes some inheritance anomalies. Method guards, standard life routines, and enable sets do best with respect to the goals. Of those three, only enable sets do not have an obvious hidden performance penality. The reflective control mechanisms are the most flexible ones, since the programmer can use them to implement other forms of coordination. However, there might be a performance penalty for the increased object consumption.




## 5. 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. Especially for distributed memory computers, network latency easily amounts to thousands of processor cycles or more. To achieve good performance, objects that are used together should preferably reside in the same memory module and activities should be assigned to processor elements that have local access to the accessed objects.

For various reasons, more than half of the languages do not consider this problem at all: Some languages have only been implemented in a prototypical way on a single workstation where network latencies do not occur; their developers have mainly been interested in the design of coordination mechanisms and a proof of concept. Other languages are restricted to shared memory multiprocessors, they rely on the cache systems provided by those machines. The remainder of the COOLs target distributed systems but do not elaborate on the strategy used to map objects and activities to the underlying machine. Those COOLs that consider the mapping problem are different in their approaches to attack it, i.e., in the way the responsibility for achieving appropriate locality is shared among compiler, run-time system, and programmer. There are two extrema.

Utmost performance and weakest portability can be achieved if the programmer hard-codes all aspects of distribution to best exploit the topology of a given machine. If the machine platform changes, a major code reworking is necessary. The following languages are in this category.

CHARM++ (PE number in index expressions)
Dragoon (assign jobs to real PEs)
Fragmented Objects, FOG++
Parallel OO Fortran (explicit placement)

At the other end of the spectrum compiler and run-time system implement heuristic strategies to achieve good locality – completely transparent for the programmer. This approach results in portable application code. However, the mapping problem is NP-complete [Mace 1987], the quality of automatic mapping strategies is debatable, especially since no comparative quantitative results are known that would allow a proper judgement of the heuristics. The following languages and compiler systems are in this category.

C** (optimized caching)
CFM (object grouping algorithm based on communication cost estimates)
CHARM++ (programmer selects from several mapping strategies)
Orca (static compiler analysis guides run-time placement decisions)
PO (mapping based on static communication analysis)
Proof (automatic object clustering)
Spar (automatic mapping for distributed arrays)

For most locality-sensitive COOLs, a solution is selected that is between both ends of the spectrum. In the remainder of this section, we discuss five common approaches for expressing locality.

A problem with all approaches is that they do not merge nicely with reuse if different modules, packages, or libraries independently specify locality for their objects. For avoiding this problem, the application programmer is often required to specify locality constraints for the whole system, including all reused parts. This either results in uninformed locality decisions for reused code or in a break of modularity, because the implementation details of reused codes are taken into account. Even more inheritance anomalies may be caused.

Another problem with all approaches is that it is almost impossible to find published performance analyses. It is unclear how good compared to explicit specification of mappings various forms of heuristics and automatic mapping functions work for collections of benchmarks and for real applications.

### 5.1 Meta-Level Locality

COOLs with reflective concurrency control (see section 4.2.6) 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.

*Discussion.* All-purpose mapping code in a meta-class results in portable but potentially slow code. Specific meta-level mapping code performs better but reflects details of both the application and the underlying topology, thus rendering the system non-portable, at least with respect to performance.

An advantage of meta-level locality is that application code and locality code can be kept separate. This allows for two-phase code development. In the first phase the application is encoded without any distribution aspects. When it is running correctly, distribution can be added and debugged in a modular way.

*Cools in this Category.*

ABCL/R2
ABCL/R3
Conc. Class Eiffel
Correlate
DROL
Java//, ProActive PDC
JavaParty
MeldC

## 5.2 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. Distribution is transparent, i.e., except for varying performance there is no difference between local and remote objects. Dynamically changing locality preferences cannot be expressed, i.e., objects do not migrate.

*Discussion.* COOL programs based on external locality are portable. The user must change the mapping of objects to processing elements when using a different topology. Although this approach is appropriate for systems with few objects, it is no longer practical for the user if many objects are involved.

*Cools in this Category.*

COOL/NTT
Java/RMI
Obliq
Procol

## 5.3 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. Except for the optional syntactic element, distribution is transparent in the code. Explicit object migration can be added by means of more optional syntactic elements.

*Discussion.* The disadvantage of internal locality is its dependence on a given machine topology. If the topology changes, all explicit processor numbers that occur in the code might require modification. Although in general there are less such places than in the purely explicit approach, bad portability remains, at least with respect to performance.

An advantage is that typically in most sections of the code automatic mapping results in sufficiently efficient code. Manual mapping can be restricted to those segments of the code that are crucial for run-time performance.

Moreover, application and distribution can be developed separately and in two phases resulting in the same advantages as in meta-level locality.

*Cools in this Category.*

ABCL/1
ABCL/f
Amber
Beta
Cool/Stanford
DOWL
Emerald
Guide
JavaParty
Mentat
Panda
POOL
Rosette
UC++

## 5.4 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.

In addition to abstract processor numbers, two other abstract machine models are known. First, data-parallel COOLs use the model of a multi-dimensional grid and express the mapping of arrays with respect to this grid. By mapping array elements of different arrays to the same position in the grid, object locality can be expressed. The same general approach is used in non-object-oriented data-parallel languages, like Fortran D [Fox et al. 1990] and HPF [Koelbel et al. 1994]. Second, scope locality offers a partitioning of the language name space into segments. 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.

*Discussion.* A virtual topology is an additional level of abstraction that enhances portability since the system automatically maps the virtual topology and the segments of the name space to the underlying machine platform. On the other hand, the programmer can guide the system to map objects to the same processor when they are used together.

The COOLs with this approach to locality do not support the same type of two-phase development as above. Instead, the distribution aspects are a central part of the total design. The reason is that most COOLs in this category do not properly decouple the mapping from the application. A later change of the mapping often results in the necessity to slightly change object access code in large fractions of the code. This is especially obvious for scope locality, because different locality is reflected in different access syntax. But similar effects can be noticed in some data-parallel COOLs as well, where local and spread array dimensions are accessed differently.

*COOLs in this Category.*

Braid (data-parallel, arrays)
comp. C++ (scope)
distrib. C++ (scope)
distrib. Eiffel (scope)
dpSather (data-parallel)
EPEE
HPJava (data-parallel, `at`, `on`)
MPC++
NAM
parallel C++
pSather (zones)
SR (scope)
Titanium (zones)

### 5.5 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 forms networks of objects that belong together. The run-time system then maps the networks to the underlying topology. Distribution is transparent. The run-time system can use transparent object migration to enhance performance.

*Discussion.* Again, a two-phase development style can be used since locality constraints can be added later. Performance is portable because object grouping is abstract and independent of the topology. Group locality has the potential of avoiding the general problem of integrating locality and object-orientation, because the programmer only expresses relations between objects that are known to him. Relationships between reused objects are specified in reused code; they transparently extend the network of object relationships.

Unfortunately, group locality as introduced so far does not adhere to the style of structured programming. Instead, locality constraints are often encoded in object attributes which are modified at various places of the user code. Therefore, it is difficult to understand and maintain the general distribution architecture of an application. Conflicting locality hints are hard to detect in large codes especially if they are caused in reused code.

*COOLs in this Category.*

Amber
A-NETL
Beta
Conc. Aggregates
COOL/Stanford
COOL/Chorus
Dowl
Emerald
Mentat

## 6. CONCLUSION

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.

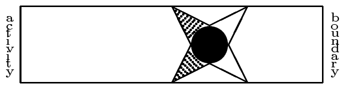
ACKNOWLEDGMENTS

Thanks to Jerome Feldman and ICSI for the support of most the research that resulted in this survey. Seth C. Goldstein helped find and structure the material. Thanks to Claudio Fleiner, Bernhard Haumacher, Lutz Prechelt, Walter Tichy, and Wolf Zimmermann for commenting on earlier drafts. An anonymous reviewer made helpful suggestions on how to improve the presentation.

## APPENDIX

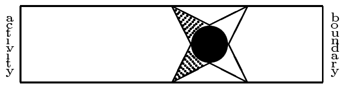
## A. LANGUAGE FEATURES

**ABCL/1** [Yonezawa 1990]




http://web.yl.is.s.u-tokyo.ac.jp
ftp://camille.is.s.u-tokyo.ac.jp
group address ⟶ abcl@is.s.u-tokyo.ac.jp
A. Yonezawa ⟶ yonezawa@is.s.u-tokyo.ac.jp

**ABCL/f** [Taura et al. 1994]

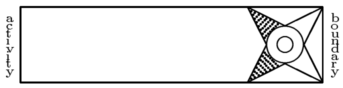



http://web.yl.is.s.u-tokyo.ac.jp
group address ⟶ abcl@is.s.u-tokyo.ac.jp
A. Yonezawa ⟶ yonezawa@is.s.u-tokyo.ac.jp

**ABCL/R2** [Masuhara et al. 1992; Yonezawa 1990]

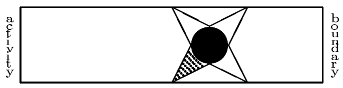

http://web.yl.is.s.u-tokyo.ac.jp
ftp://camille.is.s.u-tokyo.ac.jp
group address ⟶ abcl@is.s.u-tokyo.ac.jp
A. Yonezawa ⟶ yonezawa@is.s.u-tokyo.ac.jp

**ABCL/R3** [Masuhara et al. 1994]

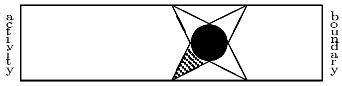



http://web.yl.is.s.u-tokyo.ac.jp
group address ⟶ abcl@is.s.u-tokyo.ac.jp
A. Yonezawa ⟶ yonezawa@is.s.u-tokyo.ac.jp

**Acore** [Manning 1988]

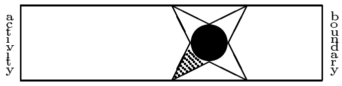


**ACT++** [Kafura 1988; Kafura and Lavender 1990; Kafura and Lee 1990; Kafura et al. 1993; Kafura and Lee 1989; Kafura and Lavender 1996]

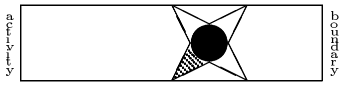



http://actor.cs.vt.edu/~kafura/act++/
Dennis Kafura ⟶ kafura@cs.vt.edu

**Act1**

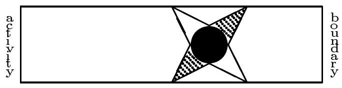



**Actalk** [Briot 1988]



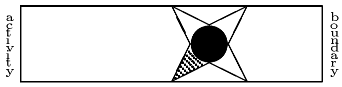

http://web.yl.is.s.u-tokyo.ac.jp/members/briot/actalk/actalk.html
ftp://ftp.ibp.fr/ibp/softs/litp/actalk
Jean-Pierre Briot ⟶ briot@is.s.u-tokyo.ac.jp

**ActorSpace** [Agha and Callsen 1993; Callsen and Agha 1994]

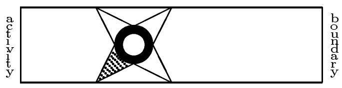



ftp://biobio.cs.uiuc.edu/pub/papers
ftp://biobio.cs.uiuc.edu/pub/theses
Christian J. Callseen ⟶ chris@iesd.auc.dk
Gul Agha ⟶ agha@cs.uiuc.edu

**Actra** [McAffer and Duimovich 1990; Thomas et al. 1988]




**Amber** [Chase et al. 1989]



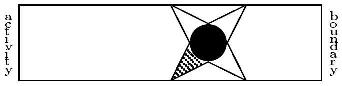

**A-NETL** [Baba 1990; Yoshinaga and Baba 1991]


http://aquila.is.utsunomiya-u.ac.jp/index_English.html

**Arche** [Benveniste and Issarny 1992]


Marc Benveniste ⟶ mbenveni@irisa.fr
Valérie Issarny ⟶ issarny@irisa.fr

**ASK** [Santo and Iannello 1990]




Guilia Iannello ⟶ iannello@udsab.dia.unisa.it

**ATOM** [Papathomas and Andersen 1997]

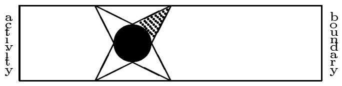

M. Papathomas ⟶ michael@comp.lancs.ac.uk

**A'UM** [Yoshida and Chikayama 1988]


**BETA** [Brandt and Madsen 1993; Madsen et al. 1993; Madsen 1993]


http://www.daimi.aau.dk/~beta
news:comp.lang.beta
http://www.mjolner.dk
information ⟶ info@mjolner.dk

**Blaze 2** [Mehrotra and Rosendale 1987; Mehrotra and Rosendale 1988]




Piyush Mehrotra ⟶ pm@icase.edu

**Braid, Data-Parallel Mentat** [West 1994; West and Grimshaw 1995]


Andrew S. Grimshaw ⟶ grimshaw@virginia.edu
group ⟶ mentat@virginia.edu

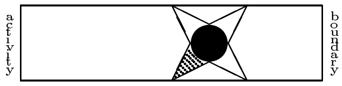
**C++//** [Caromel et al. 1996], see **Eiffel//**


Denis Caromel ⟶ caromel@mimosa.unice.fr

**C**** [Larus 1992; Larus et al. 1992; Larus et al. 1994]


http://www.cs.wisc.edu/~wwt
James R. Larus ⟶ larus@microsoft.com

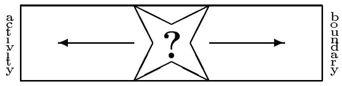
**Cantor** [Athas and Boden 1988]




**CEiffel** [Löhr 1992; Löhr 1993]


Klaus-Peter Löhr ⟶ lohr@inf.fu-berlin.de

**CFM** [Uehara and Tokoro 1990]



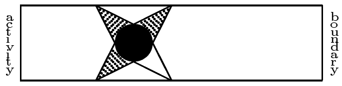

**CHARM++** [Kale and Krishnan 1993]


http://charm.cs.uiuc.edu
ftp://a.cs.uiuc.edu/pub/CK
Laxmikant V. Kale ⟶ kale@cs.uiuc.edu
Sanjeev Krishnan ⟶ sanjeev@cs.uiuc.edu

**CLIX** [Hur and Chon 1987]


**COB** [Hosokawa and Nakamura 1989]


**Compositional C++, CC++** [Carlin et al. 1993; Chandy and Kesselman 1992; Chandy and Kesselman 1993; Foster 1994]

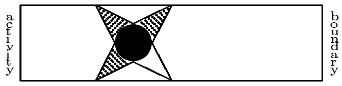



http://globus.isi.edu/ccpp
Carl Kesselman ⟶ carl@compbio.caltech.edu

**Concurrency Class for Eiffel** [Karaorman and Bruno 1993a; Karaorman and Bruno 1993b]

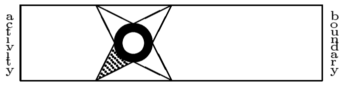

Murat Karaorman ⟶ murat@cs.ucsb.edu
John Bruno ⟶ bruno@cs.ucsb.edu

**Concurrent Aggregates, CA** [Chien 1990; Chien 1993; Chien et al. 1994; Karamcheti and Chien 1993; Plevyak et al. 1995]

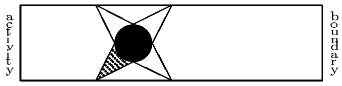



http://www-csag.ucsd.edu
Andrew A. Chien ⟶ achien@cs.ucsd.edu

**ConcurrentSmalltalk** [Yokote and Tokoro 1986]




**cooC** [Trehan et al. 1993]




ftp://isl.rdc.toshiba.co.jp/pub/toshiba
group ⟶ cooc@isl.rdc.toshiba.co.jp

**COOL (Chorus)** [Amaral et al. 1992; Lea et al. 1993; Lea and Weightman 1991]


ftp://ftp.chorus.fr/pub
news:comp.os.chorus
group ⟶ info@chorus.com

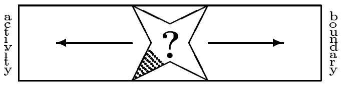
**COOL (NTT), ACOOL** [Maruyama and Raguideau 1994]


ftp://ftp.ntt.jp/pub/lang
K. Maruyama ⟶ maruyama@nttmfs.ntt.jp

**COOL (Stanford)** [Chandra et al. 1990; Chandra et al. 1993; Chandra et al. 1994]


http://www-flash.stanford.edu/cool/cool.html
Rohit Chandra ⟶ rohit@cool.stanford.edu

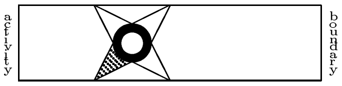
**Coral** [Chang 1990]




**Correlate** [Joosen et al. 1997]


http://www.cs.kuleuven.ac.be/∼xenoops/CORRELATE

**CST, Concurrent Smalltalk (MIT)** [Dally and Chien 1988; Horwat et al. 1989]




William Dally ⟶ dally@ai.mit.edu
Andrew Chien ⟶ achien@cs.uiuc.edu

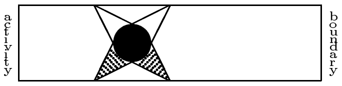
**Demeter** [Lopes and Lieberherr 1994]


http://www.ccs.neu.edu/home/lieber/demeter.html
Karl Lieberherr ⟶ lieber@ccs.neu.edu
Cristina Lopes ⟶ lopes@parc.xerox.com

**Distributed C++, DC++** [Carr et al. 1993b; Carr et al. 1993a]

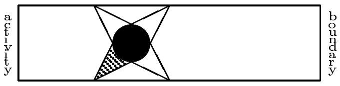

ftp://cs.utah.edu/pub/dc++
Harold Carr ⟶ carr@cs.utah.edu

**Distributed Eiffel** [Gunaseelan and LeBland 1992]




**Distributed Smalltalk – Object** [Bennet 1987; Decouchant 1986; McCullough 1987; Nascimento and Dollimore 1992; Schelvis and Bledoeg 1988]


**Distributed Smalltalk – Process** [Lee et al. 1991]




**DoPVM** [Hartley and Sunderam 1993]


ftp://mathcs.emory.edu/pub/vss
V. S. Sunderam ⟶ vss@mathcs.emory.edu
Charles Hartley ⟶ skip@mathcs.emory.edu

**DOWL, distributed Trellis/Owl** [Achauer 1993a; Achauer 1993b]


B. Achauer ⟶ bruno@tk.uni-linz.ac.at

**dpSather** [Schmidt 1992]


Schmidt ⟶ Heinz.Schmidt@fcit.monash.edu.au

**Dragoon** [Atkinson et al. 1991; Atkinson et al. 1990]

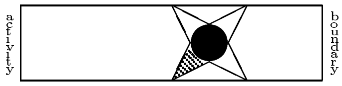

Colin Atkinson ⟶ atkinson@cl.uh.edu
Marco De Michele ⟶ demichel@txt.it

**DROL** [Takashio and Tokoro 1992]


**Eiffel//** [Caromel 1988; Caromel 1989; Caromel 1990a; Caromel 1990b; Caromel 1993; Caromel and Rebuffel 1993]


http://www-sop.inria.fr/croap/eiffel-ll
Denis Caromel ⟶ caromel@mimosa.unice.fr

**Ellie** [Andersen 1992; Andersen 1993]




ftp://ftp.diku.dk/diku/dists/ellie/papers
B. Andersen ⟶ andersen@informatik.uni-kl.de

**Emerald** [Hutchinson et al. 1987; Jul 1989; Jul et al. 1988]


ftp://ftp.diku.dk/pub/diku/dists/emerald
Eric Jul ⟶ eric@diku.dk

**EPEE, Eiffel Parallel Execution Env.** [Hamelin et al. 1994; Jézéquel 1992; Jézéquel 1993; Jézéquel 1993]


http://www.irisa.fr/EXTERNE/projet/pampa/EPEE
Jean-Marc Jézéquel ⟶ jezequel@irisa.fr

**ES-Kit Software** [Smith 1991; Tiemann 1988]


http://www.mcc.com

**ESP – Extensible Software Platform** [Chatterjee 1993]


David Croley ⟶ croley@mcc.com
Arun Chatterjee ⟶ arun@mcc.com

**Fleng++** [Tanaka 1991]


**Fragmented Objects, FOG/C++** [Gourhant and Shapiro 1990; Makpangou et al. 1993]


http://pauillac.inria.fr/cdrom_a_graver/projs/sor/eng.htm
Yvon Gourhand ⟶ gourhant@corto.inria.fr

**Guide** [Chevalier et al. 1993; Decouchant et al. 1989; Hagimont et al. 1994; Krakowiak et al. 1990; Lacourte 1991; Riveill 1992]


http://www.imag.fr
ftp://ftp.imag.fr/pub/GUIDE

**HAL** [Houck and Agha 1992; Kim and Agha 1992]


http://yangtze.cs.uiuc.edu
Gul Agha ⟶ agha@cs.uiuc.edu

**Harmony** [MacKay et al. 1988]


**Heraklit** [Hindel 1988]

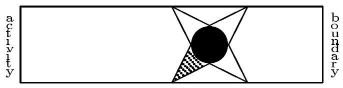

http://www2.informatik.uni-erlangen.de/IMMD-II/Research/Projects/HERAKLIT
P. Arius ⟶ arius@informatik.uni-erlangen.de
W. Betz ⟶ betz@informatik.uni-erlangen.de

**HoME** [Ogata et al. 1992]


**HPJava** [Carpenter et al. 1997; Carpenter et al. 1998]


http://www.npac.syr.edu/projects/pcrc/HPJava/

**Hybrid** [Nierstrasz 1987; Nierstrasz 1992; Papathomas 1992]

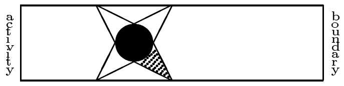



Oscar Nierstrasz ⟶ oscar@iam.unibe.ch

**IceT** [Gray and Sunderam 1997]

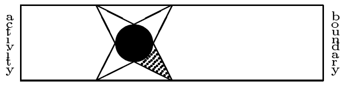

http://www.mathcs.emory.edu/icet
Paul Gray ⟶ gray@mathcs.emory.edu

**Java**




http://java.sun.com
group ⟶ java@java.sun.com

**Java's model on parallel machines**




—**Java/DSM** [Yu and Cox 1997]
—**JavaParty** [Philippsen and Zenger 1997; Philippsen and Haumacher 1998; Philippsen et al. 2000]
http://wwwipd.ira.uka.de/JavaParty/
—**Manta** [Maassen et al. 1999]
http://www.cs.vu.nl/manta
—**Do!** [Launay and Pazat 1996]

**Java//, ProActive PDC** [Caromel et al. 1998], see **Eiffel//**

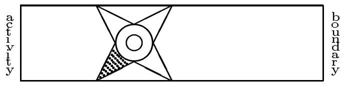

http://www.inria.fr/sloop/javall
Denis Caromel ⟶ caromel@mimosa.unice.fr

**Karos** [Guerraoui 1992]


**LO** [Andreoli et al. 1991]


**Mediators** [Grass and Campbell 1986]


**MeldC** [Kaiser et al. 1993; Kaiser et al. 1993; Popovic et al. 1990]




group ⟶ MeldC@cs.columbia.edu
Gail E. Kaiser ⟶ kaiser@cs.columbia.edu

**Mentat** [Grimshaw 1993a; Grimshaw 1993b; Grimshaw and Vivas 1991; Grimshaw et al. 1993; Grimshaw et al. 1994; Group 1995]


http://www.cs.virginia.edu/~mentat
ftp://uvacs.cs.virginia.edu
group ⟶ mentat@virginia.edu
Andrew S. Grimshaw ⟶ grimshaw@virginia.edu

**Meyer's Proposal** [Meyer 1993]


Betrand Meyer ⟶ bertrand@eiffel.com

**Micro C++, $\mu$C++** [Buhr et al. 1992; Buhr and Ditchfield 1992; Buhr and Stroobosscher 1993]


http://plg.uwaterloo.ca/~pabuhr/uC++.html
group ⟶ usystem@maytag.uwaterloo.ca

**Modula-3*** [Heinz 1993]


Ernst A. Heinz ⟶ heinze@ira.uka.de

**MPC++** [Ishikawa 1994; Ishikawa et al. 1993]


http://www.rwcp.or.jp/people/mpslab/mpc++
Yutaka Ishikawa ⟶ ishikawa@rwcp.or.jp

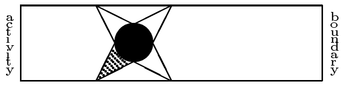
**Moose** [Waldorf and Bagrodia 1994]


http://may.cs.ucla.edu/projects/moose

**Multiprocessor Smalltalk** [Pallas and Ungar 1988]

**NAM** [Lee and Chen 1993]


**Obliq** [Cardelli 1994; Cardelli 1995]




http://www.research.digital.com/SRC/home.html
Luca Cardelli ⟶ luca@src.dec.com

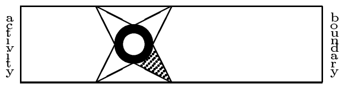
**Orca** [Bal 1991; Bal 1993; Bal et al. 1992; Bal and Kaashoek 1993; Heinzle et al. 1994; Tanenbaum et al. 1992]


http://www.cs.vu.nl/orca
Henri E. Bal ⟶ bal@cs.vu.nl

**Oz, Perdio, Distributed Oz** [Henz 1994; Smolka 1994; Smolka 1995; Smolka et al. 1995; Haridi et al. 1997]


http://www.mozart-oz.org
group ⟶ oz@dfki.uni-sb.de
Gerd Smolka ⟶ smolka@dfki.uni-sb.de

**Panda** [Assenmacher et al. 1993]




http://www.uni-kl.de/AG-Nehmer

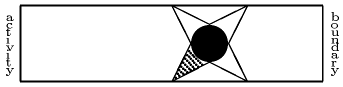
**Parallel C++, pC++** [Bodin et al. 1993; Bodin et al. 1993; Lee and Gannon 1991]


http://www.extreme.indiana.edu/sage
Dennis Gannon ⟶ gannon@cs.indiana.edu

**Parallel Computing Action** [Saleh and Gautron 1991a; Saleh and Gautron 1991b]

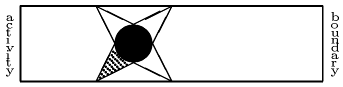

Hayssam Saleh ⟶ saleh@litp.ibp.fr
Philippe Gautron ⟶ gautron@litp.ibp.fr

**Parallel Object-Oriented Fortran** [Reese and Luke 1991]




ftp://ftp.erc.msstate.edu
Donna Reese ⟶ dreese@erc.msstate.edu

**PO** [Corradi and Leonardi 1988; Corradi et al. 1992]



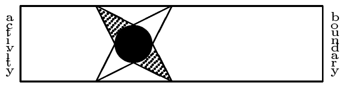

**POOL, POOL-T, POOL-I** [America 1987a; America 1987b; America 1990a; America 1990b; Spek 1990; Wester and Hulshof 1990]


**Presto** [Bershad et al. 1998; Bershad 1988]


ftp://ftp.cs.washington.edu/pub

**Procol** [Bos and Laffra 1989; Laffra and van den Bos 1990a; Laffra and van den Bos 1990b]


**Proof** [Yau et al. 1991]




**pSather** [Stoutamire 1995; Philippsen 1995a; Stoutamire 1997]


http://www.icsi.berkeley.edu/∼sather
news:comp.lang.sather
David Stoutamire ⟶ davids@icsi.berkeley.edu

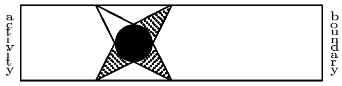
**PVM++** [Pozo 1992]


Roldan Pozo ⟶ pozo@nist.giv

**Python**


http://www.python.org

**QPC++** [Boles 1993]




D. Boles ⟶ boles@informatik.uni-oldenburg.de

**Rosette** [Tomlinson et al. 1988; Tomlinson and Singh 1989; Tomlinson et al. 1991]


ftp://biobio.cs.uiuc.edu

**SAM** [Prelle et al. 1990]

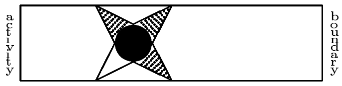

**Scheduling Predicates** [McHale 1994; McHale et al. 1991]


**Scoop** [Vaucher et al. 1988]


**Smalltalk-80** [Goldberg and Robson 1983]


**Sos** [McHale 1994]


**Spar** [van Reeuwijk et al. 1997]




http://pds.twi.tudelft.nl/projects/spar

**Synchronizing Resources, SR** [Andrews and Olsson 1993; Andrews 1981; Olsson et al. 1992]


http://www.cs.arizona.edu/sr/www
group ⟶ sr-project@cs.arizona.edu
Gregory R. Andrews ⟶ greg@cs.arizona.edu

**Titanium** [Yellick et al. 1998; Stoutamire 1997]

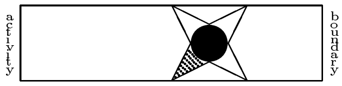

http://www.cs.berkeley.edu/projects/titanium

**Tool** [Carvalho 1993]


http://www.inf.puc-rio.br/∼sergio/tool
S. E. R. de Carvalho ⟶ sergio@inf.puc-rio.br

**Trellis/Owl** [Moss and Kohler 1987; Schaffert et al. 1986; Schaffert et al. 1985]


**Ubik** [de Jong 1990]




Peter De Jong ⟶ pdjong@vnet.ibm.com

**UC++** [Winder et al. 1992]


http://www.dcs.kcl.ac.uk/UC++
Russel Winder ⟶ russel@dcs.kcl.ac.uk

## REFERENCES


ACHAUER, B. 1993a. The DOWL distributed object-oriented language. *Communications of the ACM 36*, 9, 48–55.

ACHAUER, B. 1993b. Implementation of distributed Trellis. In *ECOOP*, LNCS 707, pp. 103–117. Springer.

AGHA, G. AND CALLSEN, C. J. 1993. ActorSpaces: An open distributed programming paradigm. In *4th ACM Symp. on Principles & Practice of Parallel Programming*, pp. 23–32.

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

AMARAL, P., JACQUEMOT, C., JENSEN, P., LEA, R., AND MIROWSKI, A. 1992. Transparent object migration in COOL2. In *ECCOP Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems*.

AMERICA, P. 1987a. Inheritance and subtyping in a parallel object-oriented language. In *ECOOP*, LNCS 276, pp. 234–242. Springer.

AMERICA, P. 1987b. POOL-T: A parallel object-oriented language. In A. YONEZAWA AND M. TOKORO Eds., *Object-Oriented Concurrent Programming*, pp. 199–220. MIT Press.

AMERICA, P. 1990a. A parallel object-oriented language with inheritance and subtyping. In *ECOOP OOPSLA*, pp. 161–168.

AMERICA, P. 1990b. POOL: Design and experience. In *ECOOP OOPSLA Workshop on Object-Based Concurrent Programming*, pp. 16–20.

ANDERSEN, B. 1992. Ellie - a general, fine-grained, first class object based language. *J. of Object-Oriented Programming 5*, 2, 35–42.

ANDERSEN, B. 1993. Efficiency by type-guided compilation. In *OOPSLA Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pp. e1–e5.

ANDREOLI, J.-M., PARESCHI, R., AND BOURGOIS, M. 1991. Dynamic programming as multiagent programming. In *ECOOP Workshop on Object-Based Concurrent Computing*, pp. 163–176.

ANDREWS, G. R. 1981. Synchronizing resources. *ACM TOPLAS 3*, 4, 405–430.

ANDREWS, G. R. AND OLSSON, R. A. 1993. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings Publishing.

ANDREWS, G. R. AND SCHNEIDER, F. B. 1983. Concepts and notations for concurrent programming. *ACM Computing Surveys 15*, 1, 3–43.

GOOS, G. AND HARTMANIS, J. Eds. 1983. *The Programming Language Ada Reference Manual.* ANSI. MIL-STD-1815A-1983.

ASSENMACHER, H., BREITBACH, T., BUHLER, P., HÜBSCH, V., AND SCHWARZ, R. 1993. PANDA – supporting distributed programming in C++. In *ECOOP*, LNCS 707, pp. 361–383. Springer.

ATHAS, W. C. AND BODEN, N. J. 1988. Cantor: An Actor programming system for scientific computing. In *SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 66–68.

ATKINSON, C., GOLDSACK, S., MAIO, A. D., AND BAYAN, R. 1991. object-oriented concurrency and distribution in DRAGOON. *J. of Object Oriented Programming 4*, 1, 11–20.

ATKINSON, C., MAIO, A. D., AND BAYAN, R. 1990. Dragoon: an object-oriented notation supporting the reuse and distribution of ada software. In *Ada Letters*, pp. 50–59.

BABA, T. 1990. A network-topology independent task allocation strategy for parallel computers. In *Supercomputing'90*, pp. 878–887.

BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. 1994. Compiler transformations for high-performance computing. *ACM Computing Surveys 26*, 4, 345–420.

BAL, H. E. 1991. A comparative study of five parallel programming languages. In *Spring Conf. on Open Distributed Systems, EurOpen*, pp. 209–228.

BAL, H. E. 1993. Comparing data synchronization in Ada9X and Orca. Technical Report IR-345, U. of Amsterdam.

BAL, H. E. AND KAASHOEK, M. F. 1993. Object distribution in Orca using compile-time and run-time techniques. In *OOPSLA*, pp. 162–177.

BAL, H. E., KAASHOEK, M. F., AND TANENBAUM, A. S. 1992. Orca: A language for parallel programming of distributed systems. *IEEE ToSE 18*, 3, 190–205.

BAL, H. E., STEINER, J. S., AND TANENBAUM, A. S. 1989. Programming languages for distributed computing systems. *ACM Computing Surveys 21*, 3, 261–322.

BANERJEE, U. 1988. *Dependence Analysis for Supercomputing.* Kluwer.

BANNING, J. P. 1979. An efficient way to find the side-effects of procedure calls and the aliases of variables. In *6th ACM Symp. on Principles of Programming Languages*, pp. 29–41.

BENNET, J. K. 1987. The design and implementation of distributed Smalltalk. In *OOPSLA*, pp. 318–330.

BENVENISTE, M. AND ISSARNY, V. 1992. Concurrent programming notations in the object-oriented language Arche. Technical Report 690, IRISA, France.

BERSHAD, B. N. 1988. The PRESTO user's manual. Technical Report 88-01-04, U. of Washington, Seattle.

BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. 1998. Presto: A system for object-oriented parallel programming. *Software – Practice and Experience 18*, 8 (August), 713–732.

BLOOM, T. 1979. Evaluating synchronization mechanisms. In *7th Symp. on Operating Systems Principles*, pp. 24–32.

BODIN, F., BECKMAN, P., GANNON, D., NARAYANA, S., AND YANG, S. X. 1993. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming 2*, 3.

BODIN, F., BECKMAN, P., GANNON, D., YANG, S. X., KESAVAN, S., MALONY, A., AND MOHR, B. 1993. Implementing a parallel C++ runtime system for scalable parallel systems. In *Supercomputing'93*, pp. 588–597.

BOLES, D. 1993. Parallel object-oriented programming with QPC++. *Structured Programming 14*, 158–172.

BORNING, A. H. 1986. Classes versus prototypes in object-oriented languages. In *ACM/IEEE Fall Joint Computer Conf.*.

BOS, J. AND LAFFRA, C. 1989. PROCOL: A parallel object language with protocols. In *OOPSLA*, pp. 95–102.

BRANDT, S. AND MADSEN, O. L. 1993. object-oriented distributed programming in BETA. In *ECOOP Workshop on Object-Based Distributed Programming*, LNCS 791, pp. 185–212. Springer.

BRIOT, J.-P. 1988. From objects to Actors: Study of a limited symbiosis in Smalltalk-80. Technical Report 88-58, Laboratoire Informatique Théorique et Programmation, Paris, France.

BRIOT, J.-P., GUERRAOUI, R., AND LÖHR, K.-P. 1998. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys 30*, 3 (September), 291–329.

BRIOT, J.-P. AND YONEZAWA, A. 1987. Inheritance and synchronization in concurrent oop. In *ECOOP*, LNCS 276, pp. 33–40. Springer.

BUHR, P. A. 1995. Are safe concurrency libraries possible? *Communications of the ACM 38*, 2, 117–120.

BUHR, P. A. AND DITCHFIELD, G. 1992. Adding concurrency to a programming language. In *USENIX C++ Technical Conf.*, pp. 207–223.

BUHR, P. A., DITCHFIELD, G., STROOBOSSCHER, R. A., YOUNGER, B. M., AND ZARNKE, C. R. 1992. μC++: concurrency in the object-oriented language C++. *Software – Practice and Experience 22*, 2, 137–172.

BUHR, P. A. AND STROOBOSSCHER, R. A. 1993. *μC++ Annotated Reference Manual, Version 3.7.* U. of Waterloo.

CALLSEN, C. J. AND AGHA, G. 1994. Open heterogeneous computing in ActorSpace. *J. of Parallel and Distributed Computing 21*, 3, 289–300.

CAMPBELL, R. H. AND HABERMANN, A. N. 1974. The specification of synchronization by path expressions. *LNCS 16*, 89–102.

CARDELLI, L. 1994. Obliq: A language with distributed scope. Technical Report 122, Digital Equipment Corporation, Systems Research Center.

CARDELLI, L. 1995. A language with distributed scope. *Computing System 8*, 1, 27–59.

CARDELLI, L. AND WEGNER, P. 1985. On understanding types, data abstractions, and polymorphism. *ACM Computing Surveys 17*, 4, 471–522.

Carlin, P., Chandy, M., and Kesselman, C. 1993. *The Compositional C++ Language Definition, Revision 0.9*. California Institute of Technology, Pasadena.

Caromel, D. 1988. A general model for concurrent and distributed object-oriented programming. In *SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 102–104.

Caromel, D. 1989. Service, asynchrony and wait-by-necessity. *J. of Object-Oriented Programming 2*, 4, 12–22.

Caromel, D. 1990a. Programming abstractions for concurrent programming. In *TOOLS Pacific*, pp. 245–253.

Caromel, D. 1990b. A solution to the explicit/implicit control dilemma. In *ECOOP OOPSLA*, pp. 21–25.

Caromel, D. 1993. Toward a method of object-oriented concurrent programming. *Communications of the ACM 36*, 9, 90–102.

Caromel, D., Belloncle, F., and Roudier, Y. 1996. *The C++// System*. MIT Press.

Caromel, D., Klauser, W., and Vayssière, J. 1998. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience 10*, 11–13, 1043–1061.

Caromel, D. and Rebuffel, M. 1993. Object-based concurrency: Ten language features to achieve reuse. In *TOOLS USA*, pp. 205–214.

Carpenter, B., Chang, Y.-J., Fox, G., Leskiw, D., and Li, X. 1997. Experiments with "HPJava". *Concurrency: Practice and Experience 9*, 6, 579–619.

Carpenter, B., Zhang, G., Fox, G., Li, X., and Wen, Y. 1998. HPJava: data parallel extensions to Java. *Concurrency: Practice and Experience 10*, 11–13, 873–877.

Carr, H., Kessler, R. R., and Swanson, M. 1993a. Compiling distributed C++. In *5th Symp. on Parallel and Distributed Processing*, pp. 496–503.

Carr, H., Kessler, R. R., and Swanson, M. 1993b. Distributed C++. *ACM SIGPLAN Notices 28*, 1, 81.

Carvalho, S. 1993. The object and event oriented language TOOL. Technical Report MCC06-93, Pontificia U., Rio de Janeiro, Brazil.

Castagna, G. 1995. Covariance and contravariance : conflict without a cause. *ACM TOPLAS 17*, 3, 431–447.

Chandra, R., Gupta, A., and Hennessy, J. L. 1990. COOL: A language for parallel programming. In *Languages and Compilers for Parallel Computing*, pp. 126–148. MIT Press.

Chandra, R., Gupta, A., and Hennessy, J. L. 1993. Data locality and load balancing in COOL. In *ACM Symp. on Principles and Practice of Parallel Programming*, pp. 249–259.

Chandra, R., Gupta, A., and Hennessy, J. L. 1994. COOL: An object-based language for parallel programming. *IEEE Computer 27*, 8, 13–26.

Chandy, K. M. and Kesselman, C. 1992. Compositional C++: Compositional parallel programming. In *5th Int. Workshop on Languages and Compilers for Parallel Computing*, Number 757 in LNCS, pp. 124–144. Springer.

Chandy, K. M. and Kesselman, C. 1993. CC++: A declarative concurrent object-oriented programming notation. In G. Agha, P. Wegner, and A. Yonezawa Eds., *Research Directions in Concurrent Object-Oriented Programming*, pp. 281–313. MIT Press.

Chang, D. T. 1990. CORAL: A concurrent object-oriented system for constructing and executing sequential, parallel and distributed applications. In *ECOOP OOPSLA Workshop on object-based concurrent programming*, pp. 26–30.

Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J. 1989. The Amber system: Parallel programming on a network of multiprocessors. Technical Report 89-04-01, U. of Washington, Seattle.

Chatterjee, A. 1993. Distributed execution of C++ programs. In *OOPSLA Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pp. b1–b6.

Cheng, D. Y. 1993. A survey of parallel programming languages and tools. Technical Report RND-93-005, NASA Ames.

Chevalier, P. Y., Freyssinet, A., Hagimont, D., Krakowiak, S., Lacourte, S., and de Pina, X. R. 1993. Experience with shared object support in the

Guide system. In *Symp. on Experiences on Distributed Systems and Multiprocessors*.

Chien, A. A. 1990. Concurrent Aggregates: Using multiple-access data abstractions to manage complexity in concurrent programs. In *ECOOP OOPSLA Workshop on object-based concurrent programming*, pp. 31–36.

Chien, A. A. 1993. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs.* MIT Press.

Chien, A. A., Karamcheti, V., Plevyak, J., and Zhang, X. 1994. *Concurrent Aggregates (CA) Language Report.* U. of Illinois at Urbana-Champaign.

Choi, J.-D., Burke, M., and Carini, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th ACM Symp. on Principles of Programming Languages*, pp. 232–245.

Conway, M. E. 1963. A multiprocessor system design. In *AFIPS Fall Joint Computer Conf.*, pp. 139–146.

Corradi, A. and Leonardi, L. 1988. PO an object model to express parallelism. In *SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 152–155.

Corradi, A., Leonardi, L., and Vigo, D. 1992. Massively parallel programming environments: How to map parallel objects on transputers. In *Transputers'92*, pp. 125–141.

Dally, W. J. and Chien, A. A. 1988. object-oriented concurrent programming in CST. In *SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 28–31.

de Jong, P. 1990. Concurrent organizational objects. In *ECOOP OOPSLA Workshop on object-based concurrent programming*, pp. 40–44.

Decouchant, D. 1986. Design of a distributed object manager for the Smalltalk-80 system. In *OOPSLA*, pp. 444–452.

Decouchant, D., Krakowiak, S., Meysembourg, M., Riveill, M., and de Pina, X. R. 1989. A synchronization mechanism for typed objects in a distributed system. In *SIGPLAN Workshop on Concurrent Object-Based Language Design*, pp. 105–107.

Dennis, J. B. and Van Horn, E. C. 1966. Programming semantics for multiprogrammed computations. *Communications of the ACM 9*, 3, 143–155.

Dijkstra, E. W. 1968a. Cooperating sequential processes. In F. Genuys Ed., *Programming Languages*. New York: Academic Press.

Dijkstra, E. W. 1968b. The structure of the 'THE' multiprogramming system. *Communications of the ACM 11*, 5, 341–346.

Dijkstra, E. W. 1975. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM 18*, 8, 453–457.

Foster, I. 1994. *Designing and Building Parallel Programs*, pp. 167–205. Addison-Wesley.

Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C.-W., and Wu, M.-Y. 1990. Fortran D language specification. Technical Report TR90079, CRPC, Rice U.

Frølund, S. 1996. *Coordinating Distributed Objects.* MIT Press.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80: The Language and Implementation.* Addison-Wesley.

Gourhant, Y. and Shapiro, M. 1990. FOG/C++: a fragmented-object generator. In *USENIX C++ Conf.*, pp. 63–74.

Grass, J. E. and Campbell, R. H. 1986. Mediators: a synchronization mechanism. In *6th Int. Conf. on Distributed Computing Systems*, pp. 468–477.

Gray, P. A. and Sunderam, V. S. 1997. IceT: Distributed computing in Java. *Concurrency: Practice and Experience 9*, 11, 1161–1167.

Grimshaw, A. S. 1993a. Easy to use object-oriented parallel programming. *IEEE Computer 26*, 5, 39–51.

Grimshaw, A. S. 1993b. The Mentat computation model – data-driven support for object-oriented parallel processing. Technical Report CS-93-30, U. of Virginia, Charlottesville.

Grimshaw, A. S. and Vivas, V. E. 1991. FALCON: A distributed scheduler for MIMD architectures. In *Symp. on Experiences with Distributed and Multiprocessor Systems*, pp. 149–163.

Grimshaw, A. S., Weissan, J. B., and Strayer, W. T. 1993. Portable run-time support for dynamic object-oriented parallel processing. Technical Report CS-93-40, U. of Virginia, Charlottesville.

Grimshaw, A. S., Weissman, J. B., West, E. A., and Loyot, Jr., E. C. 1994. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *J. of Parallel and Distributed Computing 21*, 3, 257–270.

Group, M. R. 1995. Mentat 2.5 programming language reference manual. Technical report, U. of Virginia, Charlottesville.

Guerraoui, R. 1992. Dealing with atomicity in object-based distributed systems. *OOPS Messenger 3*, 3, 10–13.

Gunaseelan, L. and LeBland, R. J. 1992. Distributed Eiffel: A language for programming multi-granular distributed objects. In *4th Int. Conf. on Computer Languages (IEEE)*.

Haddon, B. K. 1977. Nested monitor calls. *Operating Systems Review 11*, 4, 18–23.

Hagimont, D., Chevalier, P.-Y., Freyssinet, A., Krakowiak, S., Lacourte, S., Mossière, J., and de Pina, X. R. 1994. Persistent shared object support in the Guide system: Evaluation & related work. In *OOPSLA*, pp. 129–144.

Hamelin, F., Jézéquel, J.-M., and Priol, T. 1994. A multi-paradigm object oriented parallel environment. In *8th Int. Parallel Processing Symp. IPPS'94*.

Hansen, P. B. 1972. Structured multiprogramming. *Communications of the ACM 15*, 7, 574–578.

Hansen, P. B. 1973a. Concurrent programming concepts. *ACM Computing Surveys 5*, 4, 223–245.

Hansen, P. B. 1973b. *Operating System Principles*. Prentice Hall.

Haridi, S., Van Roy, P., and Smolka, G. 1997. An overview of the design of distributed Oz. In *Prod. of the 2nd Int. Symp. on Parallel Symbolic Computation, PASCO'97*, pp. 176–187.

Hartley, C. L. and Sunderam, V. S. 1993. Concurrent programming with shared objects in networked environments. In *7th Int. Parallel Processing Symp.*, pp. 471–478.

Heinz, E. A. 1993. Modula-3*: An efficiently compilable extension of Modula-3 for explicitly parallel problem-oriented programming. In *Joint Symp. on Parallel Processing*, pp. 269–276.

Heinzle, H.-P., Bal, H. E., and Langendoen, K. 1994. Implementing object-based distributed shared memory on Transputers. In *Transputer Applications and Systems '94*.

Henz, M. 1994. The Oz notation. Technical report, DFKI, German Research Center for Artificial Intelligence, Saarbrücken, Germany.

Hindel, B. 1988. An object-oriented programming language for distributed systems: HERAKLIT. In *SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 114–116.

Hoare, C. A. R. 1972. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott Eds., *Operating Systems Techniques*, pp. 61–71. New York: Academic Press.

Hoare, C. A. R. 1974. Monitors: An operating system structuring concepts. *Communications of the ACM 17*, 10, 549–557.

Horwat, W., Chien, A. A., and Dally, W. J. 1989. Experience with CST: programming and implementation. In *ACM Conf. on Programming Language Design and Implementation*, pp. 101–109.

Hosokawa, K. and Nakamura, H. 1989. Concurrent programming in COB. In *Proc. of the Japan/UK Workshop on Concurrency: Theory, Language and Architecture*, pp. 142–156.

Houck, C. and Agha, G. 1992. HAL: A high-level Actor language and its distributed implementation. In *21st Int. Conf. on Parallel Processing, ICPP '92*, Volume II, pp. 158–165.

Hur, J. H. and Chon, K. 1987. Overview of a parallel object-oriented language CLIX. In *ECOOP*, LNCS 276, pp. 265–273. Springer.

Hutchinson, N. C., Raj, R. K., Black, A. P., Levy, H. M., and Jul, E. 1987. The Emerald programming language report. Technical Report 87-10-07, U. of Washington, Seattle.

Ishikawa, Y. 1994. The MPC++ programming language v1.0 specification with commentary. Technical Report TR-94014, Tsukuba Research Center, Real World Computing Partnership, Japan.

Ishikawa, Y., Hori, A., Konaka, H., Maeda, M., and Tomokiyo, T. 1993. MPC++: A parallel programming language and its parallel objects support. In *OOPSLA Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pp. j1–j5.

Jézéquel, J.-M. 1992. EPEE: an Eiffel environment to program distributed memory parallel computers. In *ECOOP*, LNCS 615, pp. 197–212. Springer.

Jézéquel, J.-M. 1993. EPEE: an Eiffel environment to program distributed memory parallel computers. *J. of Object Oriented Programming 6*, 2, 48–54.

Jézéquel, J.-M. 1993. Transparent parallelisation through reuse: between a compiler and a library approach. In *ECOOP*, LNCS 707, pp. 384–405. Springer.

Joosen, W., Robben, B., Van Wulpen, H., and Verbaeten, P. 1997. Experiences with an object-oriented parallel language: The Correlate project. In *Proc. International Scientific Computing in Object-Oriented Parallel Environments Conf.*.

Jul, E. 1989. Migration of light-weight processes in Emerald. *IEEE Operating Sys. Technical Committee Newsletter, Special Issue on Process Migration 3*, 1, 25–30.

Jul, E., Levy, H., Hutchinson, N., and Black, A. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. on Computer Systems 6*, 1, 109–133.

Kafura, D. 1988. Concurrent object-oriented real-time systems research. In *SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 203–205.

Kafura, D. and Lavender, G. 1990. Recent progress in combining Actor based concurrency with object-oriented programming. In *ECOOP OOPSLA*, pp. 55–58.

Kafura, D. and Lee, K. H. 1990. ACT++: Building a concurrent C++ with Actors. *J. of Object Oriented Programming 3*, 1, 25–37.

Kafura, D., Mukherji, M., and Lavender, G. 1993. ACT++ 2.0: A class library for concurrent programming in C++ using Actors. *J. of Object Oriented Programming 6*, 6, 47–55.

Kafura, D. G. and Lavender, R. G. 1996. Concurrent object-oriented languages and the inheritance anomaly. In T. Casavant, P. Tvrdik, and F. Plásil Eds., *Parallel Computers: Theory and Practice*, pp. 221–264. IEEE Computer Society Press.

Kafura, D. G. and Lee, K. H. 1989. Inheritance in Actor based concurrent object-oriented languages. In *ECOOP*, pp. 131–145.

Kaiser, G. E., Hseush, W., Lee, J. C., Wu, S. F., Woo, E., Hilsdale, E., and Meyer, S. 1993. MeldC: A reflective object-oriented coordination language. Technical Report CUCS-001-93, Columbia U., New York.

Kaiser, G. E., Hseush, W., Popovich, S. S., and Wu, S. F. 1993. Multiple concurrency control policies in an object-oriented programming system. In G. Agha, P. Wegner, and A. Yonezawa Eds., *Research Directions in Concurrent Object-Oriented Programming*, pp. 195–210. MIT Press.

Kale, L. V. and Krishnan, S. 1993. Charm++: A portable concurrent object oriented system based on C++. In *OOPSLA*, pp. 91–109.

Karamcheti, V. and Chien, A. 1993. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *ACM Supercomputing'93*, pp. 598–607.

Karaorman, M. and Bruno, J. 1993a. Design and implementation issues for object-oriented concurrency. In *OOPSLA Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pp. m1–m9.

Karaorman, M. and Bruno, J. 1993b. Introduction of concurrency to a sequential language. *Communications of the ACM 37*, 9, 103–116.

Kessels, J. L. W. 1977. An alternative to event queues for synchronization in monitors. *Communications of the ACM 20*, 7, 500–503.

Kim, W. and Agha, G. 1992. Compilation of a highly parallel Actor-based language. In *5th Int. Workshop on Languages and Compilers for Parallel Computing*, LNCS 757, pp. 1–12. Springer.

Koelbel, C. H., Loveman, D. B., Schreiber, R. S., Jr., G. L. S., and Zosel, M. E. 1994. *The High Performance Fortran Handbook.* MIT Press.

Kooper, K. D. and Kennedy, K. 1989. Fast interprocedural alias analysis. In *16th ACM Symp. on Principles of Programming Languages*, pp. 49–59.

Korson, T. and McGregor, J. D. 1990. Understanding object-oriented: A unifying paradigm. *Communications of the ACM 33*, 9, 40–60.

Krakowiak, S., Meysembourg, M., Van, H. N., Riveill, M., Roisin, C., and de Pina, X. R. 1990. Design and implementation of an object-oriented, strongly typed language for distributed applications. *J. of Object Oriented Programming 3*, 3, 11–22.

Lacourte, S. 1991. Exceptions in Guide, an object-oriented language for distributed applications. In *ECOOP*, LNCS 512, pp. 268–287. Springer.

Laffra, C. and van den Bos, J. 1990a. Constraints in concurrent object-oriented environments. In *ECOOP OOPSLA Workshop on object-based concurrent programming*, pp. 64–67.

Laffra, C. and van den Bos, J. 1990b. Propagators and concurrent constraints. In *ECOOP OOPSLA Workshop on object-based concurrent programming*, pp. 68–72.

Landi, W., Ryder, B. G., and Zhang, S. 1993. Interprocedural modification side effect analysis with pointer aliasing. In *ACM Conf. on Programming Language Design and Implementation*, pp. 56–67.

Larus, J. 1992. C**: A large-grain object-oriented, data-parallel programming language. In *5th Int. Workshop on Languages and Compilers for Parallel Computing*, LNCS 757, pp. 326–341. Springer.

Larus, J. R., Richards, B., and Viswanathan, G. 1992. C**: A large-grain object-oriented, data-parallel programming language. Technical Report UWTR-1126, U. of Wisconsin, Madison.

Larus, J. R., Richards, B., and Viswanathan, G. 1994. LCM: Memory system support for parallel language implementation. In *6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 208–218.

Launay, P. and Pazat, J.-L. 1996. Integration of control and data parallelism in an object oriented language. In *Proc. of 6th Workshop on Compilers for Parallel Computers, CPC'1996*.

Lea, R., Jacquemot, C., and Pillevesse, E. 1993. COOL: System support for distributed programming. *Communications of the ACM 36*, 9, 37–46.

Lea, R. and Weightman, J. 1991. Supporting object oriented languages in an distributed environment: The COOL approach. In *TOOLS USA*.

Lee, J. K. and Chen, Y.-Y. 1993. Compiler and library support for aggregate object communications on distributed memory machines. In *OOPSLA Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pp. d1–d10.

Lee, J. K. and Gannon, D. 1991. Object oriented parallel programming – experiments and results. In *Supercomputing'91*, pp. 273–282.

Lee, Y. S., Huang, J. H., and Wang, F. J. 1991. A distributed Smalltalk based on process-object model. In *15th Int. Computer Software and Applications Conf.*, pp. 465–471.

Lister, A. 1977. The problem of nested monitor calls. *Operating Systems Review 11*, 3, 5–7.

Löhr, K.-P. 1992. Concurrency annotations. *ACM SIGPLAN Notices 27*, 10, 327–340.

Löhr, K.-P. 1993. Concurrency annotations for reusable software. *Communications of the ACM 36*, 9, 81–89.

Lopes, C. V. and Lieberherr, K. J. 1994. Abstracting process-to-function relations in concurrent object-oriented applications. In *ECOOP*, LNCS 821, pp. 81–99. Springer.

Maassen, J., van Nieuwport, R., Veldema, R., Bal, H. E., and Plaat, A. 1999. An efficient implementation of Java's remote method invocation. In *Proc. of the 7th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP*, pp. 173–182.

Mace, M. E. 1987. *Memory Storage Patterns in Parallel Processing.* Kluwer.

MacKay, S., Gentleman, W., Stewart, D., and Wein, M. 1988. Harmony as an object-oriented operating system. In *SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 209–211.

Madsen, O. L. 1993. Building abstractions for concurrent object-oriented programming. Technical report, Aarhus U., Denmark.

Madsen, O. L., Moller-Pedersen, B., and Mygaard, K. 1993. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley.

Makpangou, M., Gourhant, Y., Le Narzul, J.-P., and Shapiro, M. 1993. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal Eds., *Readings in Distributed Computing Systems*. IEEE Computer Society Press.

Manning, C. 1988. A peek at Acore, an Actor core language. In *SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 84–86.

Maruyama, K. and Raguideau, N. 1994. Concurrent object-oriented language COOL. *ACM SIGPLAN Notices 29*, 9, 105–114.

Masuhara, H., Matsuoka, S., Watanabe, T., and Yonezawa, A. 1992. object-oriented concurrent reflective languages can be implemented efficiently. In *OOPSLA*.

Masuhara, H., Matsuoka, S., and Yonezawa, A. 1994. An object-oriented concurrent reflective language for dynamic resource management in highly parallel computing. In *IPSJ SIG Notes*, pp. 57–64.

Matsuoka, S. and Yonezawa, A. 1993. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa Eds., *Research Directions in Concurrent Object-Oriented Programming*, pp. 107–150. MIT Press.

McAffer, J. and Duimovich, J. 1990. Actra – an industrial strength concurrent object oriented programming system. In *ECOOP OOPSLA Workshop on object-based concurrent programming*, pp. 82–84.

McCullough, P. L. 1987. Transparent forwarding: First steps. In *OOPSLA*, pp. 331–341.

McHale, C. 1994. Synchronization in concurrent, object-oriented languages: Expressive power, genericity and inheritance. Ph. D. thesis, Trinity College, Dublin, Ireland.

McHale, C., Walsh, B., Baker, S., and Donnelly, A. 1991. Scheduling predicates. In *ECOOP Workshop on Object-Based Concurrent Computing*, pp. 177–193.

Mehrotra, P. and Rosendale, J. V. 1987. The BLAZE language: A parallel language for scientific programming. *Parallel Computing 5*, 339–361.

Mehrotra, P. and Rosendale, J. V. 1988. Concurrent object access in BLAZE 2. In *SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 40–42.

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

Meyer, B. 1992. Applying design by contract. *IEEE Computer 25*, 10, 40–51.

Meyer, B. 1993. Systematic concurrent object-oriented programming. *Communications of the ACM 36*, 9, 56–80.

Moss, J. E. B. and Kohler, W. H. 1987. Concurrency features for the Trellis/Owl language. In *ECOOP*, LNCS 276, pp. 171–180. Springer.

Nascimento, C. and Dollimore, J. 1992. Behavior maintenance of migrating objects in a distributed object-oriented environment. *JOOP 25*, 9, 25–33.

Nierstrasz, O. 1987. Active objects in Hybrid. In *OOPSLA*, pp. 243–253.

Nierstrasz, O. 1992. A tour of Hybrid: A language for programming with active objects. In D. Mandrioli and B. Meyer Eds., *Advances in Object-Oriented Software Engineering*, pp. 167–182. Prentice Hall.

Nuttal, M. 1994. A brief survey of systems providing process or object migration facilities. *Operating Systems Review 28*, 4, 64–80.

Ogata, K., Kurihara, S., Inari, M., and Doi, N. 1992. The design and implementation of HoME. In *ACM Conf. on Programming Languages, Design and Implementation*, pp. 44–54.

Olsson, R. A., Andrews, G. R., Coffin, M. H., and Townsend, G. M. 1992. SR – a language for parallel and distributed programming. Technical Report

TR 92-09, U. of Arizona, Tucson.

PALLAS, J. AND UNGAR, D. 1988. Multiprocessor Smalltalk a case study of a multiprocessor-based programming environment. In *SIGPLAN Conf.*, pp. 268–277.

PAPATHOMAS, M. 1989. Concurrency issues in object-oriented programming languages. In D. TSICHRITZIS Ed., *Object Oriented Development*, pp. 207–245. U. of Geneva, Switzerland.

PAPATHOMAS, M. 1992. Language design rationale and semantic framework for concurrent object-oriented programming. Ph. D. thesis, U. of Geneva, Switzerland.

PAPATHOMAS, M. AND ANDERSEN, A. 1997. Concurrent object-oriented programming in Python with ATOM. In *Proceedings of the 6th Int. Python Conf.*, pp. 77–87.

PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12, 1053–1058.

PHILIPPSEN, M. 1995a. Enabling compiler transformations for pSather 1.1. Technical Report TR-95-040, Int. Computer Science Institute, Berkeley.

PHILIPPSEN, M. 1995. Imperative concurrent object-oriented languages: An annotated bibliography. Technical Report TR-95-049, Int. Computer Science Institute, Berkeley.

PHILIPPSEN, M. AND HAUMACHER, B. 1998. Locality optimization in JavaParty by means of static type analysis. In *7th Int. Workshop on Compilers for Parallel Computers, CPC'1998* (Linköping, Sweden, June 29 – July 1, 1998), pp. 34–41.

PHILIPPSEN, M., HAUMACHER, B., AND NESTER, C. 2000. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, to appear.

PHILIPPSEN, M. AND ZENGER, M. 1997. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience 9*, 11, 1225–1242.

PLEVYAK, J., ZHANG, X., AND CHIEN, A. A. 1995. Obtaining sequential efficiency for concurrent object-oriented languages. In *22nd SIGACT–SIGPLAN Symp. on Principles of Programming Languages*, pp. 311–321.

POPOVIC, S. S., KAISER, G. E., AND WU, S. F. 1990. MELDing transactions and objects. In *ECOOP OOPSLA Workshop on object-based concurrent programming*, pp. 94–98.

POZO, R. 1992. A stream-based interface in C++ for programming heterogeneous systems. In *CRNS-NSF Workshop on Environment and Tools for Parallel Scientific Computing*, pp. 162–177. Elsevier.

PRELLE, M. J., WOLLRATH, A. M., BRANDO, T. J., AND BENSLEY, E. H. 1990. The impact of selected concurrent language constructs on the SAM run-time system. In *ECOOP OOPSLA Workshop on object-based concurrent programming*, pp. 99–103.

PRESSMANN, R. S. 1987. *Software Engineering*. McGraw-Hill Book Company, New York.

REESE, D. S. AND LUKE, E. 1991. Object oriented Fortran for development of portable parallel programs. In *3rd IEEE Symp. on Parallel and Distributed Processing*, pp. 608–615.

RIVEILL, M. 1992. An overview of the Guide language. In *2nd Workshop on Objects in Large Distributed Applications*.

SALEH, H. AND GAUTRON, P. 1991a. A concurrency control mechanism for C++ objects. In *ECOOP Workshop on object-based concurrent computing*, pp. 195–210.

SALEH, H. AND GAUTRON, P. 1991b. A system library for C++ distributed applications on Transputer. In *3rd Int. Conf. on Applications of Transputers*, pp. 638–643.

SANTO, M. D. AND IANNELLO, G. 1990. Implementing actor-based primitives on distributed-memory architectures. In *ECOOP OOPSLA Workshop on object-based concurrent programming*, pp. 45–49.

SCHAFFERT, C., COOPER, T., BULLIS, B., KILIAN, M., AND WILPOLT, C. 1986. An introduction to Trellis/Owl. In *OOPSLA*, pp. 9–16.

SCHAFFERT, C., COOPER, T., AND WILPOLT, C. 1985. Trellis – object-based environment: Language reference manual. Technical Report DEC-TR-372, Eastern Research Lab, DEC, Hudson, Massachusetts.

SCHELVIS, M. AND BLEDOEG, E. 1988. The implementation of a Distributed Smalltalk. In *ECOOP*, LNCS 322, pp. 212–232. Springer.

SCHMIDT, H. W. 1992. Data parallel object-oriented programming. In *5th Australian Supercomputer Conf.*, pp. 263–272.

SMITH, R. J. 1991. Experimental systems kit – final project report. Technical report, Microelectronics and Computer Technology Corporation, MCC, Austin, Texas.

SMOLKA, G. 1994. The definition of kernel Oz. Technical report, DFKI, German Research Center for Artificial Intelligence, Saarbrücken, Germany.

SMOLKA, G. 1995. An Oz primer. Technical report, DFKI, German Research Center for Artificial Intelligence, Saarbrücken, Germany.

SMOLKA, G., HENZ, M., AND WÜRTZ, J. 1995. object-oriented concurrent constraint programming in Oz. In P. VAN HENTENRYCK AND V. SARASWAT Eds., *Principles and Practice of Constraint Programming*, pp. 27–48. MIT Press.

SNYDER, A. 1986. Encapsulation and inheritance. In *OOPSLA*, pp. 38–45.

SPEK, J. 1990. POOL-X and its implementation. In *Parallel Database Systems. PRISMA Workshop*, pp. 309–344.

STOUTAMIRE, D. 1995. The pSather 1.0 manual and specification. Technical Report, Int. Computer Science Institute, Berkeley.

STOUTAMIRE, D. 1997. Portable, modular expression of locality. Ph. D. thesis, U. of California at Berkeley. Available as ICSI technical report 97-056.

TAKASHIO, K. AND TOKORO, M. 1992. DROL: An object-oriented programming language for distributed real-time systems. In *OOPSLA*, pp. 276–294.

TANAKA, H. 1991. A parallel object oriented language FLENG++ and its control system on the parallel machine PIE64. In *Concurrency: Theory, Language and Architecture. Japan/UK Workshop Proc.*, pp. 157–172.

TANENBAUM, A. S., KAASHOEK, M. F., AND BAL, H. E. 1992. Parallel programming using shared objects and broadcasting. *IEEE Computer 25*, 18, 10–19.

TAURA, K., MATSUOKA, S., AND YONEZAWA, A. 1994. ABCL/f: A future-based polymorphic typed concurrent object-oriented language – its design and implementation. In *DIMACS workshop on Specification of Parallel Algorithms*.

THOMAS, D. A., LALONDE, W. R., DUIMOVICH, J., WILSON, M., MCAFFER, J., AND BARRY, B. 1988. Actra - a multitasking/multiprocessing Smalltalk. In *SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 87–89.

THOMAS, L. 1992. Extensibility and reuse of object-oriented synchronization components. In *Proc. Int. Conf. on Parallel Languages and Environments*, LNCS 605, pp. 261–275. Springer.

TIEMANN, M. D. 1988. Solving the RPC problem in GNU C++. Technical Report ESKIT-285-88, Microelectronics and Computer Technology Corporation, MCC, Austin, Texas.

TOMLINSON, C., KIM, W., SCHEEVEL, M., SINGH, V., WILL, B., AND AGHA, G. 1988. Rosette: an object-oriented concurrent system architecture. In *SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 91–93.

TOMLINSON, C., SCHEEVEL, M., AND SINGH, V. 1991. *Report on Rosette 1.1.* Object-Oriented and Distributed Systems Laboratory, Microelectronics and Computer Technology Corp., MCC.

TOMLINSON, C. AND SINGH, V. 1989. Inheritance and synchronization with Enabled-sets. In *OOPSLA*, pp. 103–112.

TREHAN, R., SAWASHIMA, N., MORISHITA, A., TOMODA, I., IMAI, T., AND ICHI MAEDA, K. 1993. Concurrent object oriented 'C' (cooC). *ACM SIGPLAN Notices 28*, 2, 45–52.

TURCOTTE, L. H. 1993. A survey of software environments for exploiting network computing resources. Technical report, Mississippi State U.

UEHARA, M. AND TOKORO, M. 1990. An adaptive load balancing method in the computational field model. In *ECOOP OOPSLA Workshop on object-based concurrent programming*, pp. 109–113.

VAN REEUWIJK, K., VAN GEMUND, A. J. C., AND SIPS, H. J. 1997. Spar: A programming language for semi-automatic compilation of parallel programs. *9*, 11 (November), 1193–1205.

Vaucher, J., Lapalme, G., and Malenfant, J. 1988. SCOOP – structured concurrent object-oriented prolog. In *ECOOP*, pp. 191–210.

Waldorf, J. and Bagrodia, R. 1994. Moose: A concurrent object oriented language for simulation. *Int. Journal of Computer Simulation 4*, 2, 235–257.

Wegner, P. 1987. Dimensions of object.based language design. In *OOPSLA*, pp. 168–182.

West, E. A. 1994. Combining control and data parallelism: Data parallel extensions to the Mentat programming language. Ph. D. thesis, U. of Virginia.

West, E. A. and Grimshaw, A. S. 1995. Braid: Integrating task and data parallelism. In *Frontiers'95, 5th Symp. on the Frontiers of Massively Parallel Computation*, pp. 211–219.

Wester, R. H. H. and Hulshof, B. J. A. 1990. The POOMA operating system. In *Parallel Database Systems. PRISMA Workshop*, pp. 396–323.

Wettstein, H. 1978. The problem of nested monitor calls revisited. *Operating Systems Review 12*, 1, 19–23.

Winder, R., Roberts, G., and Wei, M. 1992. CoSIDE and parallel object-oriented languages. In *Addendum to OOPSLA*, pp. 211–213.

Wolfe, M. 1989. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. Pitman, London.

Wyatt, B., Kavi, K., and Hufnagel, S. 1992. Parallelism in object-oriented languages: a survey. *IEEE Computer 11*, 6, 56–66.

Yaoqing, G. and Kwong, Y. C. 1993. A survey of implementations of concurrent, parallel and distributed Smalltalk. *ACM SIGPLAN Notices 28*, 9, 29–35.

Yau, S. S., Jia, X., Bae, D.-H., Chidambaram, M., and Oh, G. 1991. An object-oriented approach to software development for parallel processing systems. In *15th Int. Computer Software and Applications Conf.*, pp. 453–5–8.

Yellick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. 1998. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience 10*, 11–13, 825–836.

Yokote, Y. and Tokoro, M. 1986. The design and implementation of ConcurrentSmalltalk. In *OOPSLA*, pp. 331–340.

Yonezawa, A. 1990. *ABCL: An Object-Oriented Concurrent System – theory, language, programming, implementation, and application.* MIT Press.

Yoshida, K. and Chikayama, T. 1988. A'UM = stream+object+relation. In *SIGPLAN Workshop on Object-Based Concurrent Programming*, pp. 55–58.

Yoshinaga, T. and Baba, T. 1991. A parallel object-oriented language A-NETL and its programming environment. In *15th Int. Computer Software and Applications Conf.*, pp. 459–464.

Yu, W. and Cox, A. 1997. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience 9*, 11, 1213–1224.