INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Imperative Concurrent Object-Oriented Languages

Michael Philippsen*
phlipp@icsi.berkeley.edu

TR-95-050

Part I, Version 0, August 1995

Abstract

During the last decade object-oriented programming has grown from marginal influence into widespread acceptance. During the same period of time, progress on the side of hardware and networking has changed the computing environment from sequential to parallel. Multi-processor workstations are state-of-the-art.

Unnumbered proposals have been made to combine both developments. Always the prime objective was to provide the advantages of object-oriented software design at the increased power of parallel machines.

However, combining both concepts has proven itself to be a notoriously difficult task. Depending on the approach, often key characteristics of either the object-oriented paradigm or key performance factors of parallelism are sacrificed, often resulting in unsatisfactory languages.

This survey first recapitulates well-known characteristics of both the object-oriented paradigm and parallel programming, before the design space of a combination is marked out by identifying various interdependences of key concepts. The design space is then filled with data points: For proposed languages we provide brief characteristics and feature tables. Both feature tables and the comprehensive bibliography listing might help to identify open questions and to prevent re-inventions.

For "Web-Surfers" we provide a wealth of interesting addresses.

Categories and Subject Descriptors:

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

*On leave from Department of Computer Science, University of Karlsruhe, Germany

Contents

1 INTRODUCTION **1**
- 1.1 Delimitation 1
- 1.2 Structure of the Text 1
- 1.3 Basics of Object-Orientation 2
- 1.4 Running Example 3

2 INITIATE CONCURRENCY **4**
- 2.1 Automatic Parallelization 4
- 2.2 Fork, Join, and Equivalents 6
- 2.3 Cobegin 11
- 2.4 Forall, Aggregate, and Equivalents 13
- 2.5 Autonomous Code 15

3 COORDINATE CONCURRENCY **17**
- 3.1 Goals of Integration 17
- 3.2 Categories 20
- 3.3 Activity Centered Coordination 21
- 3.4 Boundary Coordination 27
  - 3.4.1 External Control 27
  - 3.4.2 Handshake Control 32
  - 3.4.3 Intermixed Handshake Control . . . 33
  - 3.4.4 Isolated Handshake Control 37
  - 3.4.5 Reflective Control 43

4 MAPPING AND LOCATION **44**

5 LANGUAGE DISCUSSION **44**
- 5.1 General Language Design Issues 44
- 5.2 Language Survey 45

6 CONCLUSION **50**

1 INTRODUCTION

During the last decade object-oriented programming has grown from marginal influence into widespread acceptance.

During the same period of time, progress on the side of hardware and networking has changed the computing environment from sequential to parallel. Currently, multi-processor workstations closely linked into a local area network are a matter of fact. Similarly, in high-performance computers the number of processors is increasing. This development can be expected to continue in future, driven by growing markets for standard processor and network technology resulting in faster processors and stronger networks at lower prices [14].

Unnumbered proposals have been made to combine both developments. For this survey we have looked at about hundred Concurrent Object-Oriented Languages (COOLs) and we are sure that there exist a lot of COOL designs which we are not aware of. Always the prime objective that resulted in the COOL design was to provide the advantages of object-oriented software design at the increased power of parallel machines.

However, combining both concepts has proven itself to be a notoriously difficult task. Depending on the approach, often key characteristics of either the object-oriented paradigm or key performance factors of parallelism are sacrificed, often resulting in unsatisfactory languages, as for example discussed in [240].

Unless the programming environment automatically extracts parallelism from a sequential implementation, the key problems a programmer faces when parallel programming are: to detect potential parallelism in an application at hand, to start the parallel activities, to coordinate their interplay, and – depending on the programming language in use – to achieve adequate performance on the available hardware platform. The first problem is beyond the scope of this survey; after a definition of the terminology used throughout this survey, we will discuss the latter three issues.

We will see for the fourth problem of mapping a parallel solution onto the underlying computing hardware there is a wide spectrum of possibilities to divide the responsibility for this task between the programmer on one the hand side and compiler and run-time system on the other hand side. The complexity often increases for languages that specifically address distributed memory systems.

Both object-oriented programming and parallel programming provide a wealth of terminology and approaches. One purpose of this survey is to define and present the used terminology. Another purpose is to help understand the dimensions that span the design space of concurrent object-oriented programming languages. The individual understanding of these dimensions will help to structure the discussion of design interdependences that drive the development of COOLs.

1.1 Delimitation

In this survey we present and discuss imperative concurrent object-oriented languages. We do not consider programming environments that are oriented towards distributed programming. The distinguishing features are interface definition languages (IDL). We skip systems if they offer such an IDL that allows different programs to use objects from a common object base. In such systems one or several programmers write programs that cooperate on shared objects. In contrast to this approach, the languages discussed in this survey are targeted to write a single program that solves a problem with explicit parallelism.

There are some survey articles on related topics. Bal discusses five parallel programming languages in [24]. Nuttal discusses systems that provide process or object migration in [184]. Cheng's contribution [72] is a collection of parallel programming languages and tools, some of which are object-oriented. Other collections are due to Turcotte [229] and Philippsen [193]. Gao and Kwong survey parallel and distributed Smalltalks in [241]. Wyatt et al. study several object-oriented languages and discuss whether the parallelism is appropriately integrated into the languages [240].

The paper by Karaorman and Bruno [138] elaborates on the design space of parallel object-oriented programming. The thesis of Papathomas [189] and an earlier paper [188] give a first classification of concurrent object-oriented languages. However, Papathomas focussed mainly on the way of combining concurrency with objects. He does not classify the broad number of languages, we look at in this report. Neither does he take more machine-oriented details into account, e.g., the way objects or processes are mapped to the underlying parallel hardware. Hence, he is not interested in migration and scheduling. His survey is slightly biased towards languages that couple concurrency to objects, instead of having the concept of threads be orthogonal to the notion of objects.

1.2 Structure of the Text

In section 2 we first discuss how concurrency commonly is initiated explicitly, before section 3 presents

means of coordination thereof. We restrict our considerations to those concepts that are relevant for concurrent object-oriented programming languages. A more detailed survey of concepts of concurrent programming can be found in [18], programming languages for distributed computing are discussed in [28].

Instead of adding citations wherever a COOL is mentioned, we postpone all citations to a cross reference table at the end of the survey in section **??**. This eases readability of the text and avoids unnecessary replications of citations, since most COOLs are mentioned more than once.

Throughout the survey, we represent certain language features with graphical symbols. These symbols help to navigate through the text. Most importantly, the graphical representation is used in the cross reference section. In that section, each language is labeled with a pictogram. The graphical elements of the pictogram serve two purposes: they can help to quickly find the corresponding explanatory text and they can be used as a basis for comparison with other languages.

1.3 Basics of Object-Orientation

Many different object-oriented languages are in use today. In this section we briefly discuss the underlying concepts. Our terminology is based on Wegner's influential article [232] and other surveying papers and text books [53, 143, 177, 200].

An **object** is the basic programming entity. It takes up a space in memory and thus has an associated address. The object stores a "state" and offers a set of routines or functions (also referred to as methods) to define meaningful operations on that state.

A language that offers objects is said to provide **data abstraction** if the state of an object can only be accessed through these routines and functions and not by directly accessing the instance variables that are used by the object to store the state. Data abstraction is sometimes called **data encapsulation**, which stresses the fact that the state of the object is guarded against external influence; since the state can only be changed by calling the offered routines; no unanticipated changes can be made by callers.

A **class** is an implementation of a set of possible objects. Objects of the same class share the same implementation. A class determines a **type**, i.e., the interface of routines and functions that are offered by that implementation. All objects of a class have the same interface, they offer the same set of routines and functions, implement the same behavior, and therefore belong to the same type. The difference between the terms **class** and **type** will be discussed below in the context of inheritance.

Languages that offer objects, but do not have the notion of classes are called **object-based** languages. Languages that offer both objects and classes are commonly referred to as **class-based** languages. Object-based language that do not offer classes but offer a mechanism to clone objects, i.e., to make several objects that adhere to a common interface and implementation are called **prototype-based** languages (see for example [40]).

The difference between classes and Ada's packages [4] is that classes determine types of the language. Objects of a class instantiate this type. Packages cannot be used to instantiate objects, but are only used to encapsulate types.

Class-based programming languages enforce a programming style which is desirable from the software engineer's point of view. Class implementations hide information regarding their internal details behind a well defined interface and hence support a modular system design [191]. A given class should be easy to replace by an alternative implementation that offers the same interface of routines and functions.

A straightforward extension of the concept of classes leads to classes that have a type argument. These **generic classes**, **container classes**, or **templates** 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 a new or more specific implementation can be made on top of existing or more general implementations. There are several common uses for the term inheritance as shown in the following diagram.

<table>
<tr><th></th><th>class</th><th>object</th></tr>
<tr><td>implementation</td><td>implementation inheritance</td><td rowspan="2">delegation</td></tr>
<tr><td>interface</td><td>interface hierarchy</td></tr>
</table>

The term inheritance usually is used with respect to classes and implementation: The implementation of a new (sub-) class is defined by extending the implementation of an existing class by just adding a new feature or by redefining and specializing the implementation

of a given routine. We call this type of inheritance **implementation inheritance**.

Two classes that do not share the same implementation and that are not derived by implementation inheritance from each other may be in an inheritance relation when the types are considered. If only the type, i.e., the interface determined by the offered routines are considered, two different classes are of the same type if they offer routines with exactly the same signatures. A class is below another class in the **interface hierarchy**, if the class offers at least the same routines. To be more exact: for all routine 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 are required for parameters and arguments. The terminology differentiates between co-variance and contravariance.

Object based inheritance is often called **delegation** [214]; see the rightmost column of the above diagram. When a routine of an object is invoked which is not explicitly provided by that object's implementation, the object delegates the call to another object from which it was derived. This ancestor then invokes the corresponding routine unless the ancestor again needs to delegate the call. Since no classes are involved, delegation can also be meaningful for object-based and prototype-based languages.

Depending on the used semantics of the term inheritance, an object-oriented language is said to offer **multiple inheritance** either if a new class can inherit from the implementations of more than one ancestor or if the 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. Although it is unusual, it can be considered to match the connotation of multiple inheritance if an object uses several other objects to delegate routine calls.

In addition to the software quality features gained by class-based languages (i.e. support for modular design and reuse of generic classes), the additional key benefits of object-orientation for software engineering are that given code can easily be extended and thus reused. The obvious way is to create a subclass of an existing class and add the specific new feature that is needed. The implementation of the existing class is (implementation) inherited without major code reworking. This is especially suitable for rapid prototyping and for application of the spiral model of software design.

The extensibility however, is only as easy as it appears on the first glance, if the following two additional characteristics are offered by the language:

Polymorphism and **dynamic binding** are other necessary characteristics of object-oriented languages. A polymorphic reference can not only refer to objects of a particular type but as well to objects of any subtype thereof. Only if polymorphism is offered can an object of a newly defined subtype be used in old code that already works for objects of the ancestor type. Polymorphism and dynamic binding are two sides of the same coin. While polymorphism allows a variable to hold objects of a type or their subtypes, dynamic binding allows that it is (conceptually) decided at runtime which code is called to execute an operation on that variable. Because it is unclear at the time the code is written which kind of object (**dynamic type**) will be used when an operation is called on a variable that can refer to objects of a specific class (**static type**) in general run-time decisions are necessary. Of course, clever compilers might be able to prove that only certain dynamic types may occur and therefore speed up the otherwise costly indirection of code selection.

Other authors, e.g. [177], claim that additional properties, e.g. automatic memory management are necessary for a language to become truly object-oriented. However, we restrict our considerations to the basic concepts mentioned above, because we think that these are sufficient to reach the quality attributes modularity, information hiding, extensibility, and support for reuse, which made object-oriented programming so popular.

1.4 Running Example

To illustrate the concepts that are presented in this survey, we will use the following example.

Consider a print server object that can be used by several clients and uses one of all available printers to print. To simplify the problem, we assume that users do not have preferred printers. The print server determines which printer to use.

The print server can be disabled. Jobs that arrive after the disabling are queued, jobs that are processed will be finished. Similarly, the print server can be enabled.

```
class PRINT_SERVER is
public interface:
       print_text(t:STRING):INT;
       disable;
```

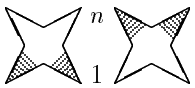
```
        enable;
end PRINT_SERVER;
```

2 INITIATE CONCURRENCY

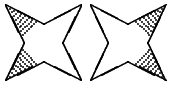
The initial question of parallel programming is how to initiate parallel execution. In this section we present various proposed approaches to make parallel execution expressible in object-oriented programming languages and discuss whether these mechanisms are appropriate in the context of object-oriented programming.

The mechanisms are categorized into five groups each of which is discussed in a subsection below. Individual subsections are labeled with a star-shaped pictogram.[1] By shading individual arms of the star we assign a special meaning to each arm.

We structure the discussion of ways to initiate parallelism along two axes. One axis deals with the number of parallel activities that can be spawned with a single language construct.



At least one of the lower two arms of the star is shaded if a language offers constructs to spawn a single new activity at a time. If one of the upper two arms is shaded then there are constructs that spawn more than one parallel activity at a time. For the discussion it is irrelevant whether the fan-out is determined at compile-time or at run-time although this of course may have performance implications.

thread-centered  object-centered

The left two arms and the right two arms reflect different understandings of parallelism. The left hand side represents a thread-centered understanding: A new activity is created and this activity has no correlation to any of the data structures in the program. On the right hand side the parallelism is inclined to a more object-centered understanding: Either the parallel activity is bound to operate on a particular data structure or it is syntactically bound to a special class or object.

None of the mechanisms discussed in the categories below is clearly superior. All of them cause some problems. Since the intended granularity of parallelism and the addressed parallel programming model cause different evaluation criteria to be applied when reasoning about these problems, existing COOLs cover all categories; most COOLs even offer mechanisms from several categories. In the remainder of this section we will present the mechanisms and discuss known drawbacks and problems that are particularly visible in context of object-oriented languages. However, we will refrain from offering a weighting and leave it to the reader to find his/her own evaluation criteria for the problems.

2.1 Automatic Parallelization

One extreme approach to parallelism is characterized by not having the programmer involved. The idea is to take a program in its sequential representation as provided by the programmer and automatically convert this program into a parallel representation which is then fed to the hardware.

We represent automatic parallelization by a star with unshaded arms to express the fact that although there is parallelism involved it is unspecified how the parallelism is initiated.

Conceptually automatic parallelization fits perfectly to object-oriented programming languages since it does not visibly interfere with existing language characteristics. However, especially in object-oriented languages automatic parallelization cannot be achieved with a sufficient degree of performance, because of the typically high number of run-time dependent object references.

Automatic parallelization can be observed in several forms which we will briefly discuss below.

Data Dependence

The key idea of an automatic transformation from a sequential to a parallel program are data dependences.

[1] The symbol was chosen because of its historic background. From the beginning of parallel programming language design the star has repeatedly been used to indicate the fact that a language is capable of expressing parallelism; for example the (non object-oriented) languages *Lisp [220], C* [221], Modula-2* [195], and the COOL C** follow this convention.

The transforming tool analyzes the given program and strives to detect sequences of operations which have no interdependences, i.e., for which it does not matter in what temporal order with respect to each other these operations are performed. If the tool can prove to itself that there are no data dependences (and of course no control dependences) then the operations can be executed in parallel.

Significant progress has been made on well defined sub-problems: Array based data dependence analysis [29, 239] and pointer or alias analysis techniques [30, 76, 142, 149] are used in various phases of compiler optimizations and to parallelize sequential code: target architectures are parallel machines, vector computers, and processors with instruction level parallelism. A good survey of the relevant techniques can be found in [23]. Everybody can convince himself of the power of these techniques by switching on the optimizing option of his favorite compiler.

For general problems however, automatic parallelization seems to be impossible, since in any given sequential program too many dependences have been introduced during the process of writing the sequential program than can be removed by any imaginary analysis tool that does not start at the problem specification itself. The limits of automatic parallelization based on dependence analysis can be shown by two examples:

- A perfect example for this are sequential implementations of sorting algorithms. No tool (except when it is based on pattern matching and hence when it is knowledgeable about a collection of problem specifications) will convert the sequential implementation in any of the most efficient parallel sorting algorithms, since except for the common problem specification sequential and parallel sorting algorithms are inherently different and often specifically tuned towards a particular architecture, see for example [35].

- Another well-known example is the performance of vectorizing compilers. If those are used on programs which have been written in the pre-vector era, the results are often poor. On more recent codes, the vectorizers do quite a good job but this is mainly due to the fact that programmers have been trained to write their programs in a vectorizable way by avoiding unnecessary dependences.

Dependence analysis has several severe limitations, especially for object-oriented programming languages because of the typically high number of run-time dependent object references. Sufficiently good automatic parallelization for general problems can only be achieved if the (imaginary) tool could perform algorithm design, i.e. if the tool could work from then problem specification and somehow derive the implementation. Since such tools are not in sight, instead of relying on automatic parallelization, parallelism is often expressed explicitly to achieve better performance. Sections 2.2 to 2.5 present categories of corresponding language constructs.

Data Flow

Data dependences are the underlying idea of programming languages based on dataflow: possibly concurrent computations are started when specific data elements are needed.

Applicative or pure functional programming languages have the key feature that their functions are free of state and do not cause any side effects. The result of a function only depends on the input arguments. In particular, there are no global variables and no pointers. If functions are free of side effects the evaluation order of their arguments is arbitrary, i.e., the order does not affect the result. For example in the expression

```
f(g(1),g(2))
```

the compiler can execute both invocations of g concurrently. Several dataflow languages are based on this principle, e.g., VAL [3] and Id90 [80].

And/Or-Parallelism

Logic programs consist of sequences of clauses as shown in the following example:

```
A :- B, C
A :- D, E
```

Each of the clauses has a declarative meaning which results from the underlying Horn logic. In the example, the declarative meaning of the first clause is "if B and C are true then A is true". Because Horn clauses have only one literal on the left hand side, an additional constructive and goal oriented procedural meaning can be defined. The procedural meaning of the first clause is "to prove that A is true, prove both subgoals B and C". At this point And-parallelism can be applied: the pure logic of Horn clauses does not prescribe any order for the proof of the subgoals B and C, hence the subgoals can be evaluated in parallel and their result must be combined by an "and"-operation. Or-parallelism can be used between clauses. In the

above example there are two different ways to prove that A is true, each of the two clauses describes one way. Hence in pure Horn logic both proofs can be tried in parallel, if either proof can be completed the goal is achieved.

Proposed logic-based object-oriented languages vary in the way in which object-oriented concepts are introduced. The most usual way is to define classes as collection of clauses which take on the role of methods. Independent of the choice, the parallelism always can be reduced to And/Or-parallelism.

COOLs in this Category

Although conceptually automatic parallelization merges fine with object-oriented languages, due to the principal restrictions of data dependence analysis the desired degree of parallelism, i.e., the desired performance cannot be achieved.

Therefore, there is no object-oriented language that solely relies on automatic parallelization. However, 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 2.2. Only Mentat and Oz use data dependence analysis to determine whether a method call can be executed concurrently. Mentat is discussed in more detail in section 7..... Above that, Mentat uses DataFlow concepts. Regular Mentat classes must be free of state, i.e., the methods of these classes are pure functions. The Mentat compiler exploits this fact to initiate parallelism automatically.

We will not discuss logic-based object-oriented languages in this survey for two reasons. First, the reader can find an excellent survey of these languages in [82]. Second, Wegner has reasoned in [233] that object-oriented and logic-programming paradigms are incompatible. We only mention Fleng++ because of its mechanism for coordination of concurrency.

2.2 Fork, Join, and Equivalents

In this section we present language constructs that start exactly one new concurrent activity at a time. This activity is not bound to objects or specific data structures of the language but can operate on the data structures in the same way as the activity which executed the construct. In the "star-notation" we shade the lower left arm of the star for languages that have such constructs.

Basic Fork and Join

The **fork** statement is the earliest proposed construct to initiate parallelism at the language level [77, 86]. Similar to a routine call, a designated routine is started with the **fork** statement. However, the invoking routine and the invoked routine proceed concurrently.

Today the **fork** statement is still very popular in thread packages (e.g. [?, 145]), where threads can be created on the fly that run in the address space of the caller. Similarly, on a more machine oriented level, operating system level processes can be forked dynamically.

Together with the **fork** statement often a **join** statement is introduced for synchronization. The process that executes the **join** statement is blocked unless/until the forked routine has terminated.

In several COOLs that are library based extensions of existing object-oriented languages, new activities are introduced in form of **thread objects**. When these objects are created, the programmer provides the name of a function and some arguments. This function is then executed concurrently. This is similar to the **fork** statement; the difference is that the **fork** statement is integrated into the language, whereas the thread object is not. The effect however, is the same, since for both constructs it is unclear at almost any point of a given code which code could be executed concurrently.

Example. In the following code, we assume that the print server ps has some devices attached, is enabled, and can handle calls of print_text.

```
err : INT;

fork(1) err1 := ps.print_text(text1) end;
fork(2) err2 := ps.print_text(text2) end;
do_some_work;
join(1);
if err1 == 0 then edit(text1) else ... end;
```

Each of the two **fork** statements spawns an additional activity that prints a text. The **fork** statement has an additional identifier that can be used in the **join** statement. While the two texts are printed, some additional work is done in do_some_work. Afterwards the **join(1)** command waits for the first print job to be completed before the text can be edited.

Discussion. By careful analysis of the above code fragment from a software engineer's point of view, the

following problem can be noticed that can even be faced in non-object-oriented languages:

- Since **fork-join** and **thread objects** did not participate in the development most programming language constructs underwent, they do not obey the single-entry-single-exit paradigm that resulted in a widespread turning away from the **goto** statement. For example, there can be several textual **join** statements that refer to a single **fork**. The programmer must make sure not to **join** dynamically more than once for a single **fork**. If two activities need synchronization, the programmer must consider all potential control flow paths to make sure to have a **join** statement in every path. Unless used with discipline the program is speckled with **fork** and **join** statements. Thus it is in general impossible to understand what routines could concurrently be executed at any point of a given program, i.e., which side effects might bother.

  To make it more obvious: when new functionality must be added to an existing **fork-join**-program the programmer must use utmost care to understand what activities might be running concurrently, and what side effects these activities could have. Since no assumptions can be made about the relative execution speeds of different activities race-conditions are frequent.

When **fork-join** and **thread objects** are offered in COOLs a key software quality characteristic that has been gained by the object-oriented paradigm is often sacrificed:

- These mechanisms may break modularity for two reasons:

  First, the caller of a method must know whether the method can be executed concurrently without any harmful interference. If this is not guaranteed by the concurrency coordination mechanism used in the COOL – and we will see in the next section that most concurrency coordination mechanisms do not offer such guarantees – the caller must study the implementation of the method to be convinced of its harmlessness. For most concurrency coordination mechanisms this necessity greatly reduces the usefulness of class libraries.

  In the example, the programmer must know that concurrent invocations of print_text do not interfere with each other. Similarly, it must be known that print_text and do_some_work do not interfere with each other. For example, the programmer must know whether the text is copied or a reference is used to access the text. In the first case the text can be changed whereas in the second case a modification of text in do_some_work could interfere with print_text. Hence, the programmer must know and understand the implementation of the PRINT_SERVER class.

  Second, it is in general impossible to change the implementation of a method without carefully analyzing all code positions that call this method. The rewritten implementation might have introduced additional coordination constraints the caller must be aware of.

  If the implementation of print_text is changed in our example, new dependences might be introduced. For instance, it might no longer be safe to invoke print_text concurrently with several texts or print_text and do_some_work might interfere. These new dependences require that the code of the caller must be checked and possibly adapted.

The following two paragraphs will discuss variations of **fork** and **join** which are most often used in COOLs to initiate concurrency. These variations essentially inherit both the advantages and disadvantages of the basic form although some advantages are gained by reducing the expressive power of the basic form.

Asynchronous Call/Message and Future

Several languages provide an asynchronous method call. This call is similar to an ordinary method call. The difference is that the called method is processed concurrently. As long as there are no return parameters, the asynchronous method call is equivalent to the **fork** statement. Note that in this case there is no **join**. The programmer cannot determine when the asynchronously called method will terminate its execution.

If however, 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: the compiler analyzes a given program and figures out when the return value really is needed. By automatic insertion of a **join**-like construct the compiler makes sure that the caller only proceeds when the result is available. This approach is implemented for example in the COOL Eiffel//. The authors coin the term **wait by necessity**. Because of the general limits of data dependence analysis, especially for object-oriented programs, this approach

is restricted to very obvious cases. In complex situations the compiler will tend to turn the asynchronous call into a synchronous one to be defensive and to ensure the intended semantics. Hence the combination of asynchronous calls and wait-by-necessity has a weaker expressive power than general **fork-join**.

Many COOLs make this dependence explicit: They introduce so-called **futures** which are equivalent to **join**-commands. A **future** is a special type of variable that has the following characteristic: After a value has been written to the future, the future behaves like an ordinary variable. The behavior is different if the future is not yet initialized. If an activity tries to read from an uninitialized future the activity is blocked until a different activity writes a value to the future. Asynchronous calls combined with futures have the same expressive power as the general **fork-join**.

There are several instantiations of **futures**. The basic futures can only hold a single value. Some COOLs extend the basic capability by defining futures as general communication buffers which implement for instance a theoretically unbound queue of result values. An extension in another direction is first-classing of futures. Basic futures are evaluated at the point of a read access. If futures are first class objects of the languages, futures themselves can be passed as parameters without forcing their evaluation. This is useful to implement delegation inheritance: If a particular method cannot provide the return value which it is supposed to write into a future, this method could pass on the future to another method which then returns a result to the original caller. Delegation inheritance is hard to implement without first class futures.

Example. In the following code fragment we use the UNIX-like &-notation to indicate an asynchronous message call:

```
err1, err2 : FUTURE INT;

err1 := ps.print_text(text1) &;
err2 := ps.print_text(text2) &;
do_some_work;
if err1 == 0 then edit(text1) else ... end;
```

The two futures err1 and err2 are used to implement **join** functionality with the two print jobs. While the two texts are printed, some additional work is done in do_some_work. When err1 is read in the **if** statement, the activity blocks until the return value of the first print job is available.

Discussion. Asynchronous method calls/messages and futures are equivalent to **fork** and **join**. The additional concept of wait-by-necessity reduces the expressive power in comparison to the basic constructs since it restricts the parallelism to those cases that can be handled by data dependence analysis. Therefore, from the point of view of the imperative or object-oriented software engineer similar problems occur:

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

- Asynchronous method calls/messages and futures may break modularity since (1) the programmer must know the implementation details of the called method and (2) the implementation can in general not be changed without an analysis and potentially a modification of the code of the caller.

Post-Processing (Early Return)

A dual approach to asynchronous method calls is post-processing. Some authors call this approach "early return". 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.

The called routine can return a result and 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. In COOLs with post-processing the programmer can return a result without terminating the method processing. Note, that the two effects are not orthogonal, since the programmer cannot terminate the method without returning a result prior to termination, if the caller expects a result.

Post-processing seems to be strange at first glance, because one usually expects a method to do some useful work that results in a return value. However, if the intention of the method is not only to produce a return value, but the focal contribution are the side effects or the changes of internal state, then post-processing is equivalent to asynchronous method calls.

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 an additional 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.

In comparison with general **fork-join** the expressive power is restricted. If one would use **fork-join** to implement post-processing behavior, **fork** statements would only be allowed at the early return commands. **Join** commands would be unnecessary.

Example. Post-processing can be used in the implementation of the PRINT_SERVER. The call of print_text returns immediately after the text is printed. However, the routine print_text continues to work and updates the accounting tables.

```
class PRINT_SERVER is
public interface:
     print_text(t:STRING):INT;
     enable;
     disable;
implementation:
     error_code : INT;
     print_text(t:STRING):INT is
          -- do the printing on some printer
          return_and_continue error_code;
          -- do the accounting
     end;
     ...
end PRINT_SERVER;
```

The following code fragment shows the code of the caller who uses the post-processing version of print_text. This code is more elegant than the versions shown before, because no future must be declared, it is irrelevant for the caller whether a method should be called synchronously or asynchronously, and there is no need for additional **join**-commands.

```
err1 := ps.print_text(text1);
err2 := ps.print_text(text2);
do_some_work;
if err1 == 0 then edit(text1) else ... end;
```

The concurrency is no longer visible in the caller. Although the accounting for both print jobs might be executed concurrently to do_some_work, the programmer can rely on the fact the the implementation of the post-processing part will take care of potential interferences. However, the expressive power is restricted: the second print job can only be started after the first one has (early) returned.

Discussion. The software engineer faces the following advantages and disadvantages:

- If only post-processing is used the parallelism is tied to class methods. Since the parallelism is restricted to the code lines after the early **return**, it is easier to derive from a program text which code could potentially be executed in parallel. This restriction eases debugging.

- If the post-processing part of a method only works on private state variables of an object or if the post-processing code makes sure that it does not interfere with any concurrently executing activity, the method can be used without a detailed understanding of implementation details. Moreover, if the implementation of such a method is changed, the caller is not affected. Hence, under the given assumptions, post-processing does not break modularity.

  Thus, in contrast to **fork** and asynchronous method calls where modularity is likely to be broken, for post-processing there exists a programming style that can avoid this adversary effect. This style could even be enforced by a language definition. The reason for this difference is that the programmer who implements the method already *knows* that the code lines in the post-processing part will be executed concurrently, which is not always the case for the other mechanisms.

- Existing (sequential) libraries can be used easily because these do not use the post-processing part. These libraries could gradually be reworked towards a parallel implementation. Such an approach is impossible for the other two mechanisms where existing sequential libraries are in general hard to use in parallel contexts.

- However, as we mentioned in the discussion of the example code, post-processing limits the expressive power. It is impossible to express the same parallelism with post-processing as with the other mechanisms.

COOLs in this Category

- **Basic Fork and Join:**

Language	Comments
Amber	thread object
Comp. C++	`spawn` command
COOL (Chorus)	fork
Demeter	thread object
Distr. C++	thread object
Distr. Smalltalk - Object	fork
Distr. Smalltalk - Process	fork
DOWL	thread object
Harmony	thread object
HoME	fork
Mediators	Life routine can `spawn` method execution. See section 2.5 for a discussion of life routines.
Multiprocessor-Smalltalk	fork
Obliq	fork, join + return value
Orca	fork
Presto	thread object
PVM++	thread object
Scoop	thread object
Smalltalk	fork
Trellis/Owl	thread object

- **Asynchronous Call/Method, Future, and Post-Processing:**

Language	Comments
ABCL/x	AS/WoW, 1st class future, caller, post
Acore	A/Wo, S/W, post
ACT++	A/WoW, 1st class future (first, last, queue), post
Act1	A/WoW, 1st class future (once, queue), post
Actalk	A/Wo, post
ActorSpace	A/Wo, post
Actra	S/WoW, post
A-NETL	AS/WoW, future, caller, post
ASK	A/Wo, S/W, post
A'UM	A/Wo
Cantor	A/Wo, post
CEiffel	AS/WoW, wait by necessity, method
CHARM++	A/Wo
CLIX	A/Wo, S/W, post
Conc. Aggregates	A/Wo, S/WoW, caller, post
Conc.Smalltalk	AS/WoW, 1st class future, caller, post
cooC	A/WoW, wait by necessity
Cool (NTT)	A/Wo, S/WoW, class
Cool (Stanford)	AS/WoW, no async. return except "event", method
Coral	A/Wo
CST	AS/WoW, future, caller
Distr. Eiffel	AS/WoW, 1st class future, caller
DROL	A/Wo, S/W, post
Ellie	AS/WoW, 1st class future, caller
ES-Kit	A/Wo, S/W, potential for manual futures
ESP	A/WoW, S/W, 1st class futures, caller
Fragmented Objects, FOG/C++	A/Wo, S/W, potential for manual futures
HAL	A/Wo, S/W, post
Heraklit	A/Wo, S/WoW, caller
Hybrid	A/Wo, S/WoW, method
Karos	A/Wo
LO	A/Wo
MeldC	A/Wo, S/WoW, caller
Mentat	AS/WoW, class, post
Meyer's Proposal	A/Wo, S/W
MPC++	A/Wo, AS/W, 1st class future, caller
Parallel Computing Action	AS/WoW, 1st class future, caller
Parallel Object-Oriented Fortran	A/Wo
PO	A/WoW, S/W, future, caller
POOL	post
Procol	A/Wo
pSather	AS/WoW, 1st class future (queue), caller
QPC++	AS/WoW, wait by necessity, caller, post
Rosette	A/Wo, S/W, post
SAM	A/Wo, post
SR	A/Wo, S/WoW, caller and method, post
Tool	S/WoW, A/Wo, class
Ubik	A/Wo, post
UC++	A/Wo, S/WoW, caller and method/class

In the above language feature table the second column provides some additional information on

how concurrency is introduced. The following paragraphs explain the used abbreviations.

Method Invocation. First, we describe the calling mode, i.e., the way methods can be invoked.

		Wo without result	*W* with result
A	asynchronous call or send message	*A/Wo*	*A/W*
S	synchronous call or call and wait	*S/Wo*	*S/W*

The upper row *(A)* shows the abbreviations for languages that provide asynchronous method calls or asynchronously sent messages. The bottom row *(S)* refers to synchronous calls, where the caller waits until the called method is executed to completion. The columns differentiate between methods without *(Wo)* or with *(W)* return value. Sequential imperative programming languages would be classified S/WoW to indicate the fact, that functions are called synchronously and that both functions without and with return value are possible.

The class A/Wo refers to languages that are based on pure message passing, i.e., a message is sent to start a method, return values are not possible. If in such a languages a return value is needed the value must be sent back by an individual explicit message.

Blocking Return Values. A/W languages need some mechanisms to indicate that the caller eventually blocks and waits for the return value, unless the called method is already finished. For the languages in this category the above language feature table gives more details on this mechanisms ("future" or "wait by necessity"): If futures are first class objects of the language, there is an entry "1st-class". An entry "queue" expresses the fact that the future can queue multiple values, a read access to the queue blocks until at least one data element is enqueued. The two entries "first" and "last" are used for futures that can be written more than once, however, only a single value is stored. In case of "first" only the value that is written first will survive; in case of "last" the last value that has been written before the read access is visible.

Choice between A and S. Some languages offer AS/W, AS/Wo, or AS/WoW, i.e., a method can be called either asynchronously or synchronously. In these languages there must be a way to express which calling mode is meant.

"Caller" refers to situations where the programmer has different forms of method calls for the synchronous and asynchronous calling mode. The entry "method" describes the dual situation in which the declaration of a method varies for methods that can be called asynchronously. The key word "class" indicates that a whole class can be specified to be used in asynchronous calls, i.e., whenever a method of an object of such a class is invoked, the call is asynchronous.

The combination of AS/W and "method" requires wait-by-necessity. Because the return value must otherwise be handled differently for synchronous and asynchronous calling modes, it is necessary to change the code of the caller if the calling mode of a method is toggled. For the same reason the combination of AS/W and "class" requires wait-by-necessity as well. COOLs that do not adhere to this requirement often disallow an alteration of the calling mode along the lines of the inheritance hierarchy. But this does not alleviate the problem when the implementation of a class is changed. Therefore, if the requirement is not met, modularity of the class implementation might be affected.

Post-processing. Finally, "post" indicates that a language offers post-processing as a means to introduce parallelism.

Note, that there are COOLs that have futures but are not mentioned in the above language table. These languages offer futures as a means of synchronization which has nothing to do with the initiation of parallelism. Activities can use these futures to achieve an ordering. The following table names these languages to avoid confusion.

Language	Comments
Conc.Class Eiffel	coordination future
Comp. C++	coordination future
Distr. C++	coordination future
Presto	coordination future

2.3 Cobegin

In this section we present a language construct that may start many concurrent activities at a time. These activities are not bound to objects or specific data structures of the language but can operate on the data structures in the same way as the activity which executed the construct. In the "star-notation" we shade the upper left arm of the star for languages that provide this construct.

Cobegin

The **cobegin** statement, which can be tracked back to Dijkstra [87] is a more 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

$$\textbf{cobegin}\ \mathsf{StmtList}_1 \mid \mathsf{StmtList}_2 \mid \ldots\ \mathsf{StmtList}_n\ \textbf{end}$$

creates n concurrently executing threads of control, each of which executes the corresponding list of statements. The essential difference to **fork-join** is the fact that the execution of the **cobegin** terminates, i.e., the original thread continues, after 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 of the created processes.

Example. The following code fragment shows the running example in a COOL that offers a **cobegin** statement.

```
cobegin
       err1 := ps.print_text(text1)
     | err2 := ps.print_text(text2)
     | do_some_work
end;
if err1 == 0 then edit(text1) else ... end;
```

As usual, the two print jobs are started concurrently, while some additional work is done in do_some_work.

Discussion. Similar to post-processing, the **cobegin** statement has advantages and disadvantages in comparison with **fork-join** and equivalent mechanisms.

- The **cobegin** statement and the **fork** statement have in common that a spawned activity has no connection to any object. Any method can be started concurrently in the body of a **cobegin** statement. But whereas the scope of a parallel execution was not easy to determine for all **fork-join** mechanisms, the **cobegin** statement restricts the scope of parallel activity to a textual portion of the program code. This eases debugging.

- COOLs with **cobegin** may break modularity because of the same reasons that have been discussed for **fork-join**, i.e., the programmer must have a detailed understanding of the code he calls and might be affected by changes in that code.[2]

COOLs in this Category

Language	Comments
ABCL/x	
Comp. C++	
Conc. Aggregates	
Guide	
Proof	
Rosette	
Scheduling	
Predicates	
SOS	
SR	co-statement

Par and Equivalents

The **par** statement is similar to the **cobegin** statement in its characteristic that the initiating activity is blocked until all activities that are spawned inside the **par** statement are terminated.

$$\textbf{par}\ \mathsf{StmtList}\ \textbf{end}$$

The difference is that StmtList is conceptually executed sequentially. The **par** statement itself does not introduce any parallelism but is used to coordinate concurrency. Languages that only offer a **par** statement, but do not have a form of the **cobegin** statement are not considered in this category of initiation of concurrency.

Only if statements are used in StmtList that initiate concurrency, the above mentioned synchronization takes place.

The **cobegin** example given above can be equivalently expressed by means of the **par** statement as shown below:

[2] Again, certain mechanisms for concurrency coordination might alleviate this problem by restricting the potential for harmful interference.

```
par
    fork (StmtList_1);
    fork (StmtList_2);
    ...
    fork (StmtList_n);
end
```

The following language feature table lists languages that have a **par** statement or an equivalent construct.

Language	Comments
COOL (Stanford)	`waitfor` statement
DOWL	Activity Set
LO	combination join
Micro C++	Block = thread boundary
pSather	`par` statement, Activity Set
Trellis/Owl	Activity Set

The programmer can explicitly add activities to an Activity Set and then wait for the completion of all those activities. Although this is similar to the **par** statement in effect 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.

2.4 Forall, Aggregate, and Equivalents

In this section we present language constructs that start possibly many concurrent activities at a time. These activities are bound to objects or specific data structures of the language, i.e., each new activity is supposed to work on a particular data element of a given data structure. In the "star-notation" we shade the upper right arm of the star for languages that provide such constructs.

Forall

Various forms of the **cobegin** statement have been introduced and found their way into parallel programming languages. Most notably, the **forall**, **doall**, and **doacross** forms:

forall i:[range] **do** StmtList(i) **end**

Here several instances of the statement list are executed concurrently, one for each element in the range.

Example. When a **forall** statement is provided, the running example might look like the code shown below. Here the print server is called for each element of an array of strings in parallel.

```
ta := ARRAY [1..n] OF STRING;
forall i:[1..n] do
     ps.print_text(ta[i])
end;
do_some_work;
```

The **forall** statement is intended for a finer granularity and is made for a higher degree of parallelism. Whereas the previously discussed mechanisms could easily express two print jobs that print from individual text variables, the **forall** version needs an array of texts (ta) to print from. Since the **forall** allows only a single statement sequence in its body, do_some_work cannot be started concurrently.[3]

Discussion. The following advantages and disadvantages can be noticed:

- Whereas **cobegin** helps the programmer in determining which processes could be executing concurrently, the various forms of **forall** statements further reduce complexity by easing the understanding of what these activities might be doing. Instead of requiring that the programmer deeply understands the behavior of n different lists of statements, the understanding of one single list is sufficient.
- The above advantage is bought by a limited expressibility. Arbitrary concurrency can either no longer be expressed or requires complicated and thus error-prone programming around the intended semantics of the language feature.
- Modularity may still be broken. Unless the language offers appropriate means for concurrency coordination, both nested **forall** statements and branching statements in the bodies make it hard to avoid race-conditions, to predict side effects, and thus to extend the functionality of a given program. The programmer still needs to completely understand the implementation of a method that is called from within a body of a **forall**. A change in this method's implementation might require that the calling code is rewritten as well. Hence, the usability of libraries is restricted.

[3] Of course there are ways around this restriction: Instead of starting n threads one could start $n+1$ and have an **if**-statement in the body of the **forall**. This **if** statement could start do_some_work for i=0 and print_text for $i > 0$. We did not show this solution because most existing implementations of **forall** statements do not allow **if** statements in their bodies.

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** which is a special form of object-based programming. Whereas in the thread-centered approach, the programmer focuses on threads and on the statements executed by them, the data-parallel programmer thinks in terms of data elements to which operations are applied in parallel. This different model of understanding alone, however, does not alleviate the coordination problem.

Aggregate

In object-oriented terminology, this type of initiation of parallelism is sometimes called **aggregate parallelism**. Languages offer a mechanism to group together several objects and then call a particular member function for all objects (sometimes for one object) 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 to apply an operation.

The following code example shows the similarity to a **forall**.

```
a ::= new_agg Element(5);
a.method
```

In the first line, a new aggregate a is created and five objects of class Element are combined in that aggregate. In the second line all objects of the aggregate call their member function method. Similar to the **forall** the programmer must be sure that concurrent executions of the member function on different objects of class Element can be executed in parallel without race-conditions.

If we assume now that a is an array with five elements, then the equivalent **forall** code looks like this:

```
forall i:[0..4] do a[i].method end
```

However, in comparison to the **forall** statement, aggregates offer a slightly restricted expressive power as the running example will show:

Example. When the language offers aggregates, the running example looks like this:

```
ta := AGGREGATE OF STRING;

ps.print_text(ta);
do_some_work;
```

As usual, print_text is invoked concurrently for all texts. Whereas for the **forall** statement a way could be found to concurrently invoke do_some_work this is no longer possible for aggregates. Thus, the additional work can only be done after the print jobs have been completed.

Discussion. As this code can be transformed into an equivalent **forall** implementation, the same disadvantages and advantages can be noticed. Moreover, aggregates have an expressive power that is even more restricted than that offered by the **forall** statement.

Variants of Aggregates

Aggregates can be defined as a high level concept of the language as shown above. Languages that are based on explicit message passing sometimes define aggregates implicitly by defining list of recipients or by linking several recipients to a single communication channel. To indicate the difference to forall and aggregate constructs we say that such languages offer a **multicast** message passing mechanism.

Another form of aggregates frequently is chosen by COOLs that are oriented towards an implementation on distributed hardware, for example a network of workstations. The special aggregate creates a single object of a given class on each node of the parallel machine. Although the similarity to replication is striking at first glance, the objects are individual and can reflect different states. Similar to the aggregates mentioned before, when a member function of this aggregate is called, this member function is invoked on all objects. Therefore, one instance of the member function executes on each node of the parallel machine. We call this type of aggregates **cluster aggregates**.

COOLs in this Category

Language	Comments
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

Language	Comments
Modula-3*	forall
NAM	data-parallel
parallel C++	data-parallel
Procol	multicast (type)
QPC++	aggregate (processor set)
SR	array of process (strip), `co`-stmt (quantifier)

2.5 Autonomous Code

In this section we present language constructs that normally 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. In the "star-notation" we shade the lower right arm of the star for languages that provide such constructs.

Process

Whereas variants of **fork-join**, **cobegin**, **forall**, and **aggregates** are control structures that initiate parallelism in the middle of an otherwise sequential program, explicit process declarations have been proposed to make parallelism more explicit. Such a process declaration resembles an ordinary variable or procedure declaration.

process P **is**
    Procedure-Body
end

Depending on the language that offers the process declaration, processes can be created statically or dynamically. Although a noteworthy difference to previously discussed mechanisms is that processes get names it is still undecided whether this is an advantage or not.

Whereas the previous mechanisms for initiating concurrency have the potential to express very fine grain parallelism, since they can express that even a single statement or method should be executed concurrently, processes are targeted towards coarse grain parallelism where a few clearly identifiable tasks must be executed together.

For the running example three processes would be needed, one for each of the concurrently executing jobs. This leads to rather awkward and complicated programs.

Although processes seem to be difficult to use in an elegant way for the running example, it should be noted that processes are quite useful for implementation of applications that show a certain communication pattern, for example applications that can be organized in client-server fashion. However, we think that a restriction to certain structures is a severe limitation of the expressive power.

The difference to the methods discussed earlier is that the parallelism is not created during the execution of an object-oriented program, i.e., the parallelism is not expressed within the object-oriented program. Instead the parallel process is a language concept that exists on top of an otherwise object-oriented language.

As well in this category are languages that allow the programmer to start separate jobs that use objects from a shared object space. We combine these languages into this category because there is exactly one activity bound to the code of the job.

Autonomous Routine

A straightforward extension of process declarations for object-oriented programming is to combine object and process declarations. When an object is created, one or several additional activities are spawned that execute specific member functions.

This approach is often taken by COOLs that are library based add-ons to object-oriented languages. Often a special class is provided in the library that has the added activity. We call the specific member function that is executed after the creation of the object an **autonomous routine**. Other COOLs allow the programmer to explicitly label one or several methods of the class implementation to be autonomous.

Whereas in some languages autonomous routines are started automatically upon object creation, other languages require an explicit call of that method.[4]

Similar to the difficulties encountered with processes, there is no elegant way to implement a concurrent execution of methods on demand without creating an additional object for each method that should be executed in parallel. Although autonomous routines are much tighter integrated into the paradigms of object-orientation than processes are, the running example still would require the creation of three addi-

[4] One could argue that the latter group of languages should be considered to provide asynchronous method calls with the calling mode declared at the method (A/Wo, method). However, the description of the COOLs in this group (Beta and Java) suggest that the autonomous routines *are* to be called (immediately) after object creation. Moreover, in both languages, these routines seem to be called only once during the life time of the object. Thus we decided to consider the group of languages here.

tional objects. Again, we consider this form of initiating concurrency to be of limited expressive power.

Life Routine

A special form of an autonomous routine is a life routine. An autonomous routine that is started automatically upon object creation is called life routine, if this routine is required for other methods to be executable. The life routine handles incoming method calls or messages that are sent to the object. If the life routine would terminate, the object could no longer be used. Life routines therefore have explicit receive statements or are called with an interrupt mechanism.

Some authors prefer to use "active" instead of "life". We do not take on this phrasing because the term "active objects" is heavily overloaded in the literature: several authors call objects without life routines "active" if they can become "active" by executing a method call concurrently. In this usage, the state of the object alters between "dormant" and "active".

Example. If the language offers a life routine, the running example looks like this:

```
class PRINT_SERVER is
public interface:
     print_text(t:STRING):INT;
     enable;
     disable;
implementation:
     life body is
          loop -- forever
               select
               [] -> receive "print_text(t)";
                     return my_print_text(t);
               [] -> receive "enable";
                     my_enable;
               [] -> receive "disable";
                     my_disable
               else
                   -- do the printing
                   -- do the accounting
               end;
          end;
     end;
     -- these routines cannot be called directly
     my_print_text(t:STRING):INT is ... end;
     my_enable is ... end;
     my_disable is ... end;
end PRINT_SERVER;
```

The **select** statement is discussed in more detail in section 3.4.4. For now, it is sufficient to know that the life routine waits until one of its branches is triggered. This happens when a method is called that is specified with the **receive** command at the beginning of each of the branches.

In the above code, the life routine is a loop that waits for the arrival of a print_text, an enable, or a disable message or method call. Upon this event, the life routine then calls the corresponding private method that implements the intended functionality.

Discussion. Let us again take on the role of the critical software engineer. The following problems should be noticed:

- One object with a life body that waits for incoming messages and then processes them in turn, does not increase the parallelism. Parallelism can be increased in several ways:

  First, the callers call asynchronously which results in the problems discussed in section 2.2.

  Second, the life routine uses one of the mechanisms described earlier in the section on initiation of parallelism. For example the statement

  return print_text(t)

  could be extended by a leading **fork**. As long as the new activities restrict their effect to local state, the same advantages can be noticed that have been stated for post-processing. However, relying on programming style does not guarantee the advantages.

  Finally, the life routine could create new objects with life routines of their own that then do the work. This seems to be advantageous because the new parallelism is encapsulated in the new objects. However, this approach contradicts object-oriented design principles, because the problem is no longer structured by objects that may find a representative in the "real world" but the design is structured procedurally, since an object is created for procedures to be called. It still remains to be seen whether this idea proves to be successful.

- It is necessary that the programmer clearly separates code that handles the interface behavior from code that implements the intended functionality. In the above example, the functionality is implemented in the last three routines, whereas the life body simply calls those. If subclasses are derived and methods are changed or added, normally the life body must be rewritten completely. If the functionality is clearly separated from the

life body, this task becomes easier. We will continue this discussion in the context of concurrency coordination below.

COOLs in this Category

Language	Comments
Arche	life routine, automatic start
Beta	autonomous routine, separate start
CEiffel	several autonomous routines, automatic start
COB	life routine, automatic start
Conc.Class Eiffel	life routine, separate start
Distr. Eiffel	process, dynamic
DoPVM	process, static
Dragoon	autonomous routine, automatic start
Eiffel//	life routine, automatic start
Emerald	autonomous routine, automatic start
Guide	process, static
Java	autonomous routine, separate start
Mediators	life routine, automatic start
Mentat	life routine, automatic start
Micro C++	autonomous and life routine, automatic start
Panda	autonomous routine, automatic start
POOL	life routine, automatic start
Proof	autonomous routine, automatic start
QPC++	life routine, automatic start
SR	autonomous and life routine, automatic start

3 COORDINATE CONCURRENCY

Language constructs that allow the coordination of concurrent activities must achieve several goals in the context of object-oriented languages. Most crucially, those constructs must offer ways to write correct programs, i.e., to orchestrate the parallelism so that different activities do not interfere with each other in unanticipated ways. Another goal is that the coordination mechanisms nicely merge with object-oriented paradigms, especially with inheritance. Finally, the mechanisms should be flexible enough to express all different forms of constraints that a parallel programmer might need.

This section is organized as follows. At first, different goals will be derived that are desirable for concurrency coordination mechanisms in object-oriented languages. Each of these goals is discussed and explained thoroughly.

Secondly, a classification scheme is presented which is based on these goals. The classification scheme is then used to organize the presentation of concurrency coordination constructs that have been proposed in existing COOLs. Each of the constructs is presented and discussed with respect to the goals and the degree to which the goals are met.

Unfortunately, to our knowledge no COOL has been proposed so far that completely fulfills all goals. Most COOL designers did not assign highest priority to a perfect integration of concurrency coordination with object-oriented paradigms. Often other design decisions seem to have been more important. The intention of this survey is not to blame existing COOLs for dirty integration of concurrency coordination with object-oriented paradigms. Instead the survey should help to identify advantages and weaknesses of existing concepts to increase the understanding of all the interdependences that are involved when this integration is done.

3.1 Goals of Integration

Goal of Callee-Oriented Coordination

It is crucial for the correctness of parallel programs, that the concurrent activities do not interfere in ways that result in erroneous program behavior.

One of the simplest race-conditions that might occur in case of lacking coordination is caused by the concurrent execution of the following statement:

$$\mathsf{a} := \mathsf{a} + 1$$

Assume that m activities execute the statement concurrently. The execution of the statement requires three elementary steps: to read the value of variable a, to increase the value by one, and to write the new value into the memory cell associated with a. Depending on the temporal interleaving, the value of the variable a can increase by any of the values in $[1..m]$. Since in general this result is not intended, and hence incorrect, better coordination of the concurrent activities is needed.

Usually these errors are hard to detect because race-conditions can cause non-deterministic erratic behavior. This non-determinism makes debugging difficult, because the presense of a debugger or an enabled trace file output can easily change the behavior of the

temporal interleaving and thus hide the error. The workshop [190] gives an overview on current research on debugging of parallel programs.

To reason about the correctness of sequential object-oriented programs usually Meyer's principle of "design by contract" [177, 178] is applied. A class C is called *locally correct*

- if after instantiation of a new object of class C, the class invariant Inv_{C} holds and
- if after execution of a method m of class C both the class invariant Inv_{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 point of the invocation. This is expressed with the following implication:

$$Inv_{\mathsf{C}} \wedge Pre_{\mathsf{m},\mathsf{C}} \xrightarrow{\mathsf{m}} Inv_{\mathsf{C}} \wedge Post_{\mathsf{m},\mathsf{C}} \qquad (1)$$

Although this definition is easy to apply to sequential object-oriented programming, it does not extend to concurrent programming. To see this, assume that implication (1) holds for all methods of a class C. Since the implication 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. Therefore, even if the class implementation is locally correct in terms of the sequential definition, the implementation might result in incorrect code when used concurrently.

Based on this observation, the following conclusions can be derived:

If a COOL requires that the caller of a method makes sure that the called methods work correctly in a parallel application, it is impossible to reason about the correctness of the implementation based on a local analysis of the implementation of individual classes. Hence, the caller must have knowledge about the implementation details of the classes that are used. This prevents modular system design which requires that implementation details of modules are hidden behind a well-defined interface, e.g., the class interface. Libraries cannot be used with the same generality one is used to from sequential programming because their correctness is hard to guarantee if the caller is required to coordinate concurrent accesses.

This leads to the first goal for a good integration of concurrency coordination and object-orientation:

Goal of Callee-Oriented Coordination: To allow for correct concurrent object-oriented programming, the concurrency coordination must be implemented at the side of the callee, i.e., in the class that is concurrently accessed.

Goal of Coordination Expressibility

If the first goal is met, it is still unclear how the definition of local correctness could be applied. Since in general it is too complex to analyze all potential interleavings of all methods of a class, the concurrency coordination constructs must allow to specify exactly those subsets of methods that fulfill implication (1) for all possible interleavings of the methods that are in such a subset.

A straightforward way to specify this is to simply disallow that several methods of a class can be executed concurrently. If only one method can be executed at a time, there are no run-time interleavings of the methods of a class. Therefore, the program text of each individual method can be analyzed locally to check whether implication (1) holds. The condition for local correctness which is used for sequential object-oriented programming can be used without any alterations.

Although this brute force approach eases correctness considerations significantly, it severely limits potential concurrency. For example, the well-known reader-writer situation cannot be expressed. For a class that offers methods that do not change the internal state of their objects, there is not only no reason for prohibiting their concurrent execution but instead, such a prohibition restricts potential parallelism and thus performance. Especially, since it is quite easy to show that for all potential interleavings of side-effect free methods, the class invariant cannot be affected and hence, implication (1) still holds.

We therefore conclude:

Goal of Coordination Expressibility: A concurrency coordination construct must allow several types of conditions to be expressible:

a) Intra-Object Concurrency. It must be possible to invoke one or several methods of an object concurrently. The concurrency coordination construct must provide means to express whether and which methods can be executed concurrently.

Now consider a situation where a method is being executed on an object. During that time, another method of that object is called which is not in the subset of methods that can be executed concurrently with the call being served. In this case, the newly arrived call is delayed until the first method call is completed. Similar delay situation can even occur if no method is being served at all. In the running example it might be

desirable that incoming print_text calls are queued if the print server is temporarily disabled or if too many print jobs are in line than can be handled.

Such conditions cannot easily be expressed with pre-conditions since a failure to fulfill a pre-condition is considered to be a fatal error. Moreover, there is no need to express such conditions in form of boolean guards attached to methods as it is standard practice for pre- and post-conditions. We call the new conditions **proceed-criteria** to stress the differences. A call of a method can only proceed if the corresponding proceed-criteria are evaluated to true. Otherwise the call is delayed until the proceed-criteria become true.

Concurrency coordination constructs should offer means to express the following additional types of proceed-criteria:

> **b) State Proceed-Criteria.** The concurrency coordination construct must provide means to express whether a method call can proceed or must be delayed because of a certain condition of the internal state of the object.
>
> **c) History Proceed-Criteria.** In addition there must be a way to express that a method call can proceed or must be delayed because of the history of earlier method calls processed by the object.

The above expressibility requirement (c) is subsumed by (b) because it is always possible to add object attributes and to keep track of the calling history in these attributes. However, since such conditions are needed quite frequently in existing concurrent object-oriented code and since the additional object attributes often both obscure the method code and make inheritance more difficult as we will see below, we decided to make history proceed-criteria a separate subgoal.[5]

Although we mentioned the three types of conditions individually, we will address then as proceed-criteria for the remainder of this discussion.

The main difference between pre-conditions and proceed-criteria is that the former are not meant to be actually evaluated during program execution. Pre-conditions might be helpful during the implementation phase but they are no longer necessary in correctly designed (and debugged) code. In contrast, proceed-criteria must conceptually be checked at each method invocation.[6]

This observation requires that conclusive definition of concurrency coordination mechanisms must address the problem that concurrently executed methods could affect the result of the evaluation of proceed-criteria. Unfortunately, many proposed mechanisms do not exactly specify how this problem is solved.

In addition, pre-conditions and proceed-criteria are different with respect to inheritance. If subclasses are derived the invariants of class and subclass must be more general or more specific depending on the understanding of subtyping and polymorphism used in a given language. Similarly pre- and post-conditions of class and subclass must imply their counterparts along or against the lines of the inheritance hierarchy, depending on whether the subclass conforms to or specializes the original class or whether the classes are in a co-variance or contra-variance relation [92]. Pre- and post-conditions are determined by the algorithms and the abstract data type offered by the class or subclass. Different implementations of the same class in general have the same invariants, pre- and post-conditions.

In contrast, proceed-criteria represent coordination constraints that are caused by the chosen implementation. They are often independent of the functionality of a class. These criteria might change, if a different implementation can be found that implements the same methods with tighter or loosened concurrency constraints. For example, it does not affect the local correctness of the print server implementation if calls of print_text are queued at a disabled printer.

Because of these differences between proceed-criteria and pre- and post-conditions inheritance of proceed-criteria is an issue in COOLs. The next two goals which are only relevant if a coordination construct fulfills the goal of callee-oriented coordination, address the problem of inheritance of proceed-criteria.

Goal of Isolated Coordination Code

Several mechanisms have been proposed to express proceed-criteria. These mechanisms are different with respect to where the concurrency coordination code is implemented in the class. Except for those mechanisms that do not have any explicit coordination code and for those mechanisms that factor coordination code out into meta-class implementations, the remain-

[5] The same thought can be used to justify that subgoal (a) is mentioned individually although conditions of type (a) could as well be expressed with state proceed-criteria.

[6] Clever compilers and programmers may be able to assert that proceed-criteria are always fulfilled for a given program. But since we are interested in the general characteristics of concurrency coordination mechanisms, we ignore potential optimizations here although they might be crucial for achieving adequate performance.

ing mechanisms either mix coordination code into the method code, i.e., into the code that actually implements the functionality, or the coordination code is part of the class interface, i.e., it is specified separately from the methods that implement functionality.

Goal of Isolated Coordination Code: A concurrency coordination mechanisms must clearly separate code that implements method functionality from code that expresses concurrency coordination constraints.

If this goal is not fulfilled, the concurrency coordination mechanisms sacrifices inheritance. If subclasses are derived, it might be necessary to re-program methods with basically unchanged functionality only because the code that affects the proceed-criteria must be changed. Such changes might snowball up and down the inheritance hierarchy and require code changes in other classes as well. However, if methods that implement class functionality are separated from coordination code, the necessary changes can often be limited. This effect has been detected by several researchers, e.g. [8, 43, 133, 188, 225]. To our knowledge the most detailed analysis of the problem is due to Matsuoka and Yonezawa [170] who initially coined the term **inheritance anomaly**.

Along the lines of the discussion of the mechanisms presented below, we will provide examples that will stress the fact that inheritance anomaly is a serious problem for the integration of concurrency coordination mechanisms and object-oriented languages.

Goal of Separable Coordination Code

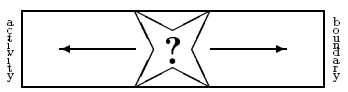
If the coordination code is isolated, i.e., if it is an isolated part of the class definition instead of being an integral part of methods that implement the functionality of the class, it is much easier to inherit from such a class. However, a significant problem is not jet addressed. There are concurrency coordination mechanisms that specify the coordination constraints for the whole class in a centralized way instead of a decentralized specification at the method level. This is a disadvantage. If a subclass has different coordination constraints it is often necessary to re-program the complete coordination code. Since often only slight changes in the coordination code are required it is better if only the affected portions of the coordination code must be changed while the remaining coordination code can be inherited.

Goal of Separable Coordination Code: A concurrency coordination mechanisms must allow to inherit portions of the coordination code separately.

3.2 Categories

There is a wide spectrum of possibilities to achieve coordination of concurrent activities. We call one end of the spectrum **activity centered coordination**. At this end the activities make sure that access to shared data is properly coordinated to avoid race-conditions. In general such mechanisms do not fulfill the goal of callee-oriented coordination. The available public interface of a class remains unchanged and is accessible for the whole life-time of an object.

We call the other end of the spectrum of possibilities **boundary coordination**. Those mechanisms fulfill the goal of callee-oriented coordination, i.e., the class implementation makes sure that methods can only be executed concurrently if their interleaving does not affect the correctness. Boundary coordination mechanisms vary greatly with respect to their expressibility. The central idea of these mechanisms is that the accessible public interface of an object is no longer considered to be static. Instead, the mechanisms allow to express that method calls must be delayed under certain conditions, which can be understood as temporarily removing a method from the public object interface.



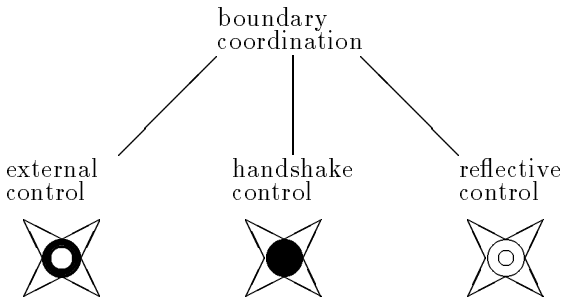
To graphically represent the spectrum we chose a slidebar. If the star that represents the COOL is on the left hand side of this bar, then the COOL provides constructs for activity centered coordination. On the right hand side of the bar, boundary coordination is used to ensure the intended behavior of concurrent activities.

In the remainder of this section we will first discuss mechanisms for activity centered coordination in section 3.3: We will present the coordination mechanisms and discuss why it is problematic not to fulfill the goal of callee-oriented coordination. Moreover, it will be shown that activity centered coordination does not fulfill the other goals either.

Since there are very different proposals for boundary coordination that are particularly relevant for object-oriented programming, we further refine the star-shaped pictogram.

activity centered

The simple circled star is used for activity centered forms of coordination. The circle is modified for different approaches of boundary coordination.



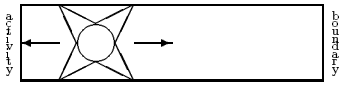
All forms of boundary coordination achieve coordination of concurrent activities at the boundary of an object. Different approaches result from a different division of responsibility between the run-time system and the object. The basic question is, where the coordination code is placed.

If there is no explicit coordination code but the language defines for all classes whether and which of concurrent method invocations will be executed, the boundary coordination mechanism is called **external control**. Section 3.4.1 discusses several approaches to external control. Although mechanisms in this group are of limited expressibility, they perfectly fulfill both goals of isolated and separable coordination code.

Similarly, in **reflective control** mechanisms the implementation of a class is again free of any coordination code. In contrast to external control there is coordination code, namely in meta-classes and hence outside of the class that is concurrently accessed. The programmer can provide coordination code in a meta-class that is used whenever coordination is required. The characteristics of languages in this category are discussed in section 3.4.5. Mechanisms in this group offer isolated coordination code; they are different with respect to expressibility and separability.

Boundary coordination mechanisms based on **handshake control** (see section 3.4.2), have explicit coordination code in the class implementation. None of the proposed mechanisms we are aware of offers isolated and separable coordination code with complete expressibility, i.e., always at least one of the goals is at least partially ignored.

3.3 Activity Centered Coordination



In this section we present and discuss several mechanisms for activity centered coordination of concurrent activities. In the graphical notation, languages that offer these mechanisms are represented by a circled star on the left hand side of the coordination slidebar. The arms of the star are shown unshaded here since the form of concurrency initiation is irrelevant for the discussion of coordination.

A general rule is that languages which have data structures that are shared and can be concurrently accessed by several activities provide mechanisms for activity centered coordination.

Mechanisms in this group in general do not fulfill the goal of callee-oriented coordination.

Synchronization by Termination

Most of the mechanisms to initiate parallelism that we have presented in section 2 provide an additional and very simple way of synchronization. Except for autonomous code (see section 2.5) the other constructs (**fork-join**, **cobegin**, **forall** and **aggregate**) provide a mechanism by which one activity can be blocked until the other (or others) have terminated. If the underlying implementation of the run-time system can efficiently handle the termination and initiation of activities, this synchronization by termination can be used by the programmer to deal with arbitrary data dependences. However, in practice **fork-join** and **cobegin** programs rely on other additional means of cooperation which are discussed below.

The **data-parallel** and **aggregate** programming approach not only provide such a mechanism but are based on this form of synchronization. Consider for example the following simple 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 an array a. Afterwards the activities set "their" element of an array b to the value stored in the array element of the array a of the "neighboring" activity.

The concurrency is coordinated by termination: After each assignment the concurrent activities terminate.[7] Therefore, activities from the second **forall** cannot interfere with activities from the first **forall**.

Example. The running example with synchronization by termination can be found in section 2.2 (**fork-join** and asynchronous call) and in section 2.3 (**cobegin**).

Discussion.

Goal ratings	
callee-oriented	no
expressive	intra-object concurrency, state proceed-criteria
isolated	n/a
separable	n/a

In combination with object-oriented languages synchronization by termination has some drawbacks. The main problems are caused by not fulfilling the goal of callee-oriented coordination.

To show this, we reconsider the aggregate example from section 2.4.

```
a ::= new_agg Element(5);
a.method
```

The above code works fine, as long as concurrent invocations of method on each of the members of the aggregate do not interfere with each other. The object-oriented paradigm requires that it should be easy to replace the implementation of a given class as long as the interface is not changed. If however, the implementation of method in class Element is changed, the programmer must make sure that concurrent invocations of the new implementation do not interfere with each other. There would have been no need to do this if method was not called from an aggregate. Thus, the way a method is used influences what is considered to be a correct implementation. This breaks the principle of modular design.

[7] Due to the nature of this example, the compiler can optimize this code: the access to a[i-1] in the second **forall** can be replaced by i-1. This allows the combination of both assignments in a single **forall** which avoids a costly synchronization barrier [194, 228]. But for sake of simplicity, let us assume that the compiler does access a[i-1] here.

A similar effect can be noticed if class Element is used to construct a derived class by implementation inheritance. If the programmer adds a new member function it is his/her task to make sure that this new function either is not called concurrently to method or does not interfere. Again usage patterns impose constraints on the class implementation.

Hence, although synchronization by termination is quite easy to understand it does not perfectly blend with concepts of object-orientation. Synchronization by termination makes it difficult to reason about the correctness of class implementations. The notion of local correctness cannot be used. Since the concurrency coordination code is not placed inside of a concurrently used class, the goals of isolated and separable coordination code cannot be fulfilled.

Since the caller can implement arbitrarily complex conditions around the **forall** statements, this concurrency coordination mechanisms offers good expressibility: It allows for intra-object parallelism since several methods of an object can be executed at the same time. Moreover, the mechanism allows the expression of state proceed-criteria. However, there is no special support for history proceed-criteria. If such criteria are needed they must manually be encoded into attributes.

Semaphore, Mutex, Lock

The second basic concept of organizing concurrent access to shared data is the semaphore, which again can be tracked back to Dijkstra [87, 88]. A semaphore is a non-negative integer variable with two atomic operations. One operation, commonly called P or wait , blocks until the variable is greater than zero in which case the variable is decreased atomically. The other operation, usually called V or signal, increases the variable atomically. When the value of the semaphore variable is greater than 1, more than one activity can pass.

The programmer must surround a critical section of the code, i.e., a section of the code that operates on shared data, by a pair of these operations.

Discussion.

Goal ratings	
callee-oriented	no, but could be yes
expressive	intra-object parallelism
isolated	no
separable	no

As we have discussed before, the goal of callee-oriented coordination is not fulfilled if the caller of methods is

responsible for the implementation of acceptable interleavings. In this case, the same problems that we have discussed for synchronization by termination are caused by semaphore, mutex, and lock. Therefore, in general modularity is broken and it is difficult to reason about the correctness of class implementations and libraries.

Conceptually however, there is a way around this problem. If the COOL design makes sure that the constructs can only be used with private class attributes, then the coordination code must be implemented in a callee-oriented way. Unfortunately, to our knowledge all COOLs that offer semaphores, mutex, or locks do not make this restriction and hence do not fulfill the goal of callee-oriented coordination.

Semaphores offer restricted expressibility. Since the initialization of the semaphore variable can allow several concurrent threads inside of the critical section, it is straightforward how to express intra-object parallelism. Semaphores do not offer any support for history proceed-conditions. Moreover, it is difficult to implement conditional coordination because the condition and its evaluation must explicitly be mapped to semaphore operations. Hence, we judge that semaphores do not offer real support for state proceed-criteria.

Semaphores do not fulfill the goal of isolated coordination code. Instead, they can be used to construct the following example that motivates the importance of this goal in object-oriented contexts.

Example. The following implementation of the print server uses semaphores only in a callee-oriented way, i.e., all semaphores are attributes of the object to which concurrent access must be coordinated. The semaphores make sure that several print_text methods can be called concurrently, but only if the print_server is enabled. Above that, the semaphores ensure that at the same time, only either enable and disable can be executed.

```
class PRINT_SERVER is
public interface:
     print_text(t:STRING):INT;
     enable;
     disable;
implementation:
     error_code : INT;
     is_enabled : SEMAPHORE := 0;
     en_dis : SEMAPHORE := 1;
     print_text(t:STRING):INT is
          is_enabled.P; is_enabled.V;
          -- do the printing
          -- do the accounting
          return error_code;
     end;
     enable:PID is
          en_dis.P;
          ... is_enabled.V; ...
          en_dis.V;
     end;
     disable is
          en_dis.P;
          ... is_enabled.P; ...
          en_dis.V;
     end;
end PRINT_SERVER;
```

The example uses two semaphores. The semaphore en_dis which is initialized to 1 makes sure that either enable or disable can be executed at a time. The other semaphore reflects the pre-condition that the print server must be enabled before use. These semaphores implement proceed-criteria because all method calls are started but may be blocked/delayed at the semaphore operation immediately after their start.

Since the semaphore operations are part of the method bodies, the coordination code is not clearly isolated. The resulting difficulties with inheritance become obvious when a subclass is derived that defines the following two new methods:

```
disable_printer(printer:PID)
enable_printer(printer:PID)
```

These routines can be used similar to the general disable/enable but affect only a single printer that is used by the print server. When defining the subclass, the implementations for print_text and disable can not be inherited, although the functionality of these methods is not changed. Although only the coordination constraints are changed, the code must be duplicated, and the new constraints must be added.

```
class PRINT_SERVER_SUB is
inherit:
     PRINT_SERVER;
public interface:
     disable_printer(printer:PID);
     enable_printer(printer:PID);
implementation:
     en_dis_printer : SEMAPHORE := 1;
     disable_printer(printer:PID) is
          is_enabled.P; en_dis_printer.P;
          ...
          en_dis_printer.V; is_enabled.V;
```

```
    end;
    enable_printer(printer:PID) is
         is_enabled.P; en_dis_printer.P;
         ...
         en_dis_printer.V; is_enabled.V;
    end;
    print_text(t:STRING):INT is
         is_enabled.P; is_enabled.V;
         en_dis_printer.P;
         -- select printer
         en_dis_printer.V;
         -- do the printing
         -- do the accounting
         return error_code;
    end;
    disable is
         en_dis_printer.P;
         en_dis.P;
         ... is_enabled.P; ...
         en_dis.V;
         en_dis_printer.V;
    end;
end PRINT_SERVER_SUB;
```

The new routines can only be executed when the print server has been enabled. This is ensured by enclosing the bodies of both routines in semaphore operations on is_enabled. While one of the two routines is executing, the other cannot be executed. For this purpose the new semaphore en_dis_printer is used. The additional coordination constraint is that print_text cannot select a printer when printers are being enabled or disabled. To implement this the code of print_text must be changed. Similarly, it does not make much sense to allow the general disable to be called while the state of individual printers is modified. Hence, the implementation of method disable must also be changed.

A second problem is a result of the first one. For instance, let us later on decide to change the implementation of PRINT_SERVER to allow only a certain number of concurrent print jobs. For this purpose we have to change the implementation of print_text in the original class. This requires an analogous change in the implementation of print_text in the derived class PRINT_SERVER_SUB.

The fundamental reason for these problems is that code that has the sole purpose of concurrency coordination is mixed with code that implements functionality. (See a similar discussion for life routines in section 2.5, where we stressed the fact that the programmer should clearly separate both issues.)

Discussion. Semaphores are a very general mechanism. Similar to the wide applicability of **fork-join**, higher-level concepts can be implemented based on them. Semaphores certainly have their place in high-performance system level implementation, however they cannot be smoothly integrated into object-oriented languages since too many goals of a good integration are not fulfilled.

Moreover, semaphores often cause the following additional software engineering problems, most of which result from the fact that semaphore operations do not obey the single-entry-single-exit paradigm.

- Access to shared data and hence surrounding semaphore operations are in general scattered throughout the code. Unless specifically documented and carefully used, it is hard to conceive the correspondence between semaphores and data elements to be shielded. Because of its flexibility a semaphore might be used several times and for various purposes in a program, rendering the program hard to maintain.

- The programmer can easily forget to close the critical section by calling the appropriate operation of the semaphore. Typical errors often involve **return**, **break**, and **goto** statements. Exceptions are particularly hard to get right in combination with semaphores.

- Semaphores are well suited for mutual exclusion. For more general conditional synchronization, semaphores require that the programmer maps the condition to semaphore, i.e., whenever a condition changes, the programmer must modify the corresponding semaphore. This can easily be overseen and results in bugs that are hard to find.

- Using several semaphores may easily lead to deadlock situations unless the programmer has a total order of the semaphores in mind and uses this order for both nested entries into and nested exits from critical sections.

  In the above code the order of the semaphore operations for example in the new routine enable_printer is crucial. If two P-operations are exchanged, a deadlock can occur if print_text or disable are called concurrently.

Higher-level constructs to coordinate concurrent activities have been invented to ease the above problems. Some of these are presented below. Although all of them can be reduced to semaphore implementations the increased level of abstraction enhances parallel programmability.

Mutex and **Lock** are special types of semaphores that allow exactly one activity to enter a critical section. Both mutex and locks can easily be implemented with semaphore operations.

Conditional Critical Region

The basic idea of conditional critical regions is to provide some syntactic support for conditional coordination of parallelism [107, 108, 115]. Whereas in the critical region defined by semaphores arbitrary code could be executed, and hence accesses to arbitrary sets of data elements could be coordinated, conditional critical regions make the purpose of coordinating of 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** $v_1, v_2, \ldots, v_m$;

The critical region is then made explicitly visible in the code by

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

StmtList is executed by an activity when the Condition holds. Otherwise or if this or another region of the resource r is concurrently being worked on by another activity, the activity blocks until the condition holds and the resource is free. Only accesses to variables in resource r are coordinated.

Example. The running example using conditional regions is shown below. The implementation defines two resources, namely enable and set_enable. The latter is used to coordinate accesses to the boolean variable is_enabled.

```
class PRINT_SERVER is
public interface:
     print_text(t:STRING):INT;
     enable;
     disable;
resources:
     resource enabled is;
     resource set_enabled is is_enabled;
implementation:
     is_enabled : BOOL;
     print_text(t:STRING):INT is
          region enabled
          when is_enabled then end;
          -- do the printing
          -- do the accounting
     end;
     enable is
          region set_enabled
          when not is_enabled then
               ... is_enabled := true; ...
          end;
     end;
     disable is
          region set_enabled
          when is_enabled then
               ... is_enabled := false; ...
          end;
     end;
end PRINT_SERVER;
```

In the above code a peculiar version of a critical region is used in the first line of the implementation of method print_text. The resource is empty, but the condition is_enabled makes sure that an activity can only proceed if the print server is in state "enabled". Otherwise the activity will block in front of the (empty) region. The code of print_text cannot be put into the body of the critical region since then only one activity could execute the code at a time.

If a subclass PRINT_SERVER_SUB is defined and the same two routines disable_printer and enable_printer are added as before, again the implementation of print_text and disable cannot be inherited but must be adapted to the new coordination constraints. Again, a change to the base implementation (e.g., allowing only a certain number of concurrent print jobs) must be re-programmed identically in the subclasses.

Discussion.

Goal ratings	
callee-oriented	no, but could easily be yes
expressive	state proceed-criteria
isolated	no
separable	no

Conditional critical regions in general do not fulfill the goal of callee-oriented coordination. However, if the COOL design restricts the scope of a resource to the class definition and allows only class attributes to be part of the resource, conditional critical regions can easily be made a mechanism for callee-oriented coordination. Unfortunately, we know of no COOL that offers conditional critical regions with this restriction.

Because of the close relationship to semaphores the expressibility is not changed significantly. The **when** clause of the **region** statement makes the implementation of state proceed-criteria much easier than it has been the case for semaphores. A disadvantage is that

only one activity can enter a region at a time. In contrast, semaphores can be used to allow several activities to pass into a portion of the code. Therefore, conditional critical regions can no longer express intra-object parallelism.

As has been discussed with the above example, the goal of isolated coordination code is not met. Consequentially, the goal of separable coordination code is not met either.

In comparison with semaphores, conditional critical regions solve a few of the commonly faced software engineering problems introduced by the generality of semaphores: Is is impossible to forget to close a critical region and it is much easier to implement higher level concepts of coordination since the conditions are explicit instead of being coded into semaphore operations.

However, more hard problems remain. It is still easy to run into deadlocks, the code that works on variables from a particular resource is still spread over the whole code, and the hard questions of what and when to encapsulate in a region must still be solved solely by the programmer.

Piggy-Backed Synchronization

In pure message passing languages, concurrently executing activities are often synchronized by blocking communication commands. Consider the following statement, where an activity waits until a message m arrives. Even a specific sender s might be specified.

receive m [**from** s]

A similar situation can be noticed, if a **call back** approach is used. Instead of an explicit **receive** statement to accept an incoming message, the recipient provides a method that is called by s. In both cases the response of the receiving activity is coordinated with respect to the sending activity. The synchronization is piggy-backed on top of the communication.

Discussion.

Goal ratings	
callee-oriented	no
expressive	state proceed-criteria
isolated	n/a
separable	n/a

Most COOLs with asynchronous method calls, messages, and futures that are discussed in section 2.2 offer some form of piggy-backed synchronization. Instead of repeating those languages in the following language feature table, we would like to identify two groups of languages that offer this type of coordination:

- languages that have (at least) calling mode A/W and futures and
- languages that have calling mode A/Wo only.

Piggy-backed synchronization always requires that the callers know the exact calling protocol. The caller must know which message a recipient might be waiting for. Since this information is not visible at the interface of the called object, the caller must know some implementation details of the called object. Hence, piggy-backed synchronization does not fulfill the goal of callee-oriented coordination. Therefore the goals of isolated and separable coordination code cannot be fulfilled either.

Moreover, piggy-backed synchronization mechanisms can only accept a single message at a time. Therefore, these constructs cannot be used to express intra-object parallelism. Only state proceed-criteria can be implemented. For this purpose the whole expressive power of the base language can be used. Since piggy-backed synchronization can be modeled by means of semaphores similar problems as the ones discussed above frequently occur in parallel programs that are based on this form of concurrency coordination.

COOLs in this Category

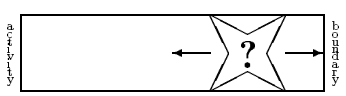
The following language feature table lists languages that are based on activity centered coordination, except for languages with piggy-backed synchronization, which can be found in the feature table of section 2.2.

Language	Comments
Amber	lock, barrier
Beta	semaphore
Blaze-2	termination (forall), lock
Braid	termination (forall)
C**	termination (forall)
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

Language	Comments
DoPVM	lock
DOWL	lock
dpSather	termination (forall)
EPEE	termination (forall)
ES-Kit	lock
Harmony	semaphore
HoME	semaphore
Java	mutex
Karos	termination
LO	termination
MeldC	mutex, semaphore
Modula-3*	termination (forall)
MPC++	mutex
Multiprocessor-Smalltalk	semaphore
NAM	termination (forall)
Obliq	mutex, lock
Panda	semaphore
parallel C++	termination (forall)
PO	semaphore
Presto	lock, mutex, coordination future
Proof	lock
pSather	lock
PVM++	lock, semaphore
QPC++	semaphore
Scoop	piggy backed sync.
Smalltalk	semaphore
SR	semaphore
Trellis/Owl	lock

Note, that several COOLs in this category, for example pSather, intentionally accept the problems of activity centered coordination in favor of different language design goals, e.g., to allow for utmost performance.

3.4 Boundary Coordination



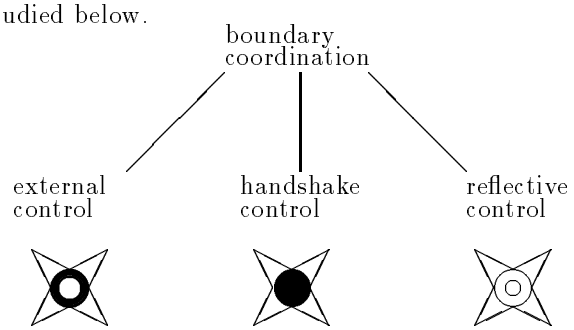
In this section we present and discuss several mechanisms for boundary coordination of concurrent activities. In the graphical notation, languages that offer these mechanisms are represented by a circled star on the right hand side of the coordination slide-bar. Depending on the type of coordination, the circle is refined. The arms of the star are unshaded here since the form of concurrency initiation is irrelevant for the discussion of coordination.

A general rule is that languages that are based on the data encapsulation paradigm can offer mechanisms for boundary coordination. If no such mechanisms are provided, forms of activity centered coordination are required instead.

Mechanisms in this group in general fulfill the goal of callee-oriented coordination.

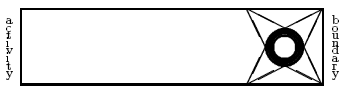
All forms of boundary coordination achieve coordination of concurrent activities at the boundary of an object. Different approaches result from a different division of responsibility between the run-time system and the object. The basic question is, where the coordination code is placed.

We group mechanisms for boundary coordination into the following three categories each of which is studied below.



There are some COOLs that offer both activity centered coordination and boundary coordination. In these cases the graphical representation depicts one of the refined circled stars on the "activity side" of the slide-bar. If a COOL offers only boundary coordination mechanisms the refined star is on the "boundary side".

3.4.1 External Control



COOLs based on boundary coordination with external control define for all classes whether and which of concurrent method invocations will be executed. Such COOLs make the run-time system responsible for proper coordination. There is no explicit concurrency coordination code that a programmer could write. An object can neither influence which of the method calls that concurrently arrive are executed nor in what order. The control is thus external, i.e., not inside of the object.

In the graphical representation the circled star has a fat rim to symbolize that control is external.

Discussion. Mechanisms in this group fulfill the goal of callee-oriented coordination. Since there is no explicit coordination code, the goals of isolated and separable coordination code are trivially met. Only the expressibility goal is not completely fulfilled. Although all constructs in this group are different with respect to their expressive power, they have the following restriction in common: All external control mechanisms allow only one method to be executed on an object at any time.

The key advantage of this restriction is that the definition of local correctness can be used without alterations. It is thus in general not more difficult to implement correct classes than it is in sequential object-oriented languages.

However, this advantage does not come for free. We have seen in the discussion of automatic parallelization in section 2.1 that the key restrictions of parallel processing result from data dependences. External concurrency control however assumes the ubiquitous existence of dependences and hence allows only one activity to execute at a time, which is in general more restrictive than appropriate for achieving parallel performance. For example, consider the classical reader-writer synchronization. Of course the desire must be to allow concurrent readers. With boundary coordination constructs based on external control this cannot be implemented.

Monitor

Although external control is frequently used in COOLs, the constellation has not been invented especially for COOLs. In fact, external control can be identified as an application of the classic monitor.

From the point of view of a parallel programmer monitors [87, 109, 116] are a syntactic combination of resource definition and the operations that can be applied to the variables in the resource. Only one operation can be executed at a time. In object-oriented terminology, a monitor resembles an object that provides data abstraction, i.e., it has internal variables to implement its state and offers procedures to the outside to perform operations on that state. The methods access the state under a mutual exclusion regime.

Discussion.

Goal ratings	
callee-oriented	yes
expressive	no
isolated	yes
separable	yes

Although monitors fulfill most of the goals of a smooth integration of concurrency control into object-oriented languages, monitors lack expressive power. It is a central part of the definition of the monitor that only one method can be active at a time. The resulting advantages and disadvantages have been addressed above.

In addition, monitors cannot be used to express conditional coordination, i.e., there is no way to express state (and hence history) proceed-criteria within the concept.

The following variant of the running example shows the consequences that will occur when conditional coordination is needed.

Example. Monitors do not offer any support for conditional coordination, i.e., it cannot be expressed that a certain method must temporarily be delayed until a certain condition holds.

One way to program around this restriction is to make the condition accessible from the outside. To achieve this in the running example, PRINT_SERVER must have an additional method

is_enabled:**BOOL**

in its interface that the caller can use to check whether a call can be made. In addition to that, the methods enable and disable are extended to have a return value. This return value tells the caller whether the method call was legal with respect to the coordination semantics. Similarly, the return value of print_text is extended to cover the case of an illegal call.

```
class PRINT_SERVER is
public interface:
     print_text(t:STRING): INT;
     enable:INT;
     disable:INT;
     is_enabled:BOOL;
implementation:
     enabled : BOOL := false;
     is_enabled:BOOL is
           return enabled;
     end;
     print_text(t:STRING):INT is
           if not enabled then return ERR
           else
                ... print and account ...
           end;
     end;
     enable:INT is
           if enabled then return ERR
           else
```

```
            ... enabled := true; ...
        end;
    end;
    disable:INT is
        if not enabled then return ERR
        else
            ... enabled := false; ...
        end;
    end;
end PRINT_SERVER;
```

Based on these new return values and the state-inquiry method the caller is in charge of implementing an appropriate behavior, i.e., to make sure that the right methods are called at the right time. This attempt to add state proceed-criteria to the monitor concept results in an unanticipated sacrifice of the goal of callee-oriented coordination.

Of course, by using two layers of monitors the problem can be solved: the outer layer offers one monitor for each condition. If the condition holds, this monitor calls the operation of the inside monitor, otherwise the outer monitor keeps re-checking the condition. This solution is not acceptable either because it leads to awkward programs that weaken the modularity of the original design.

Another disadvantage that monitors inherit from the underlying semaphore concept is that careless usage can still result in deadlocks. The problem is often referred to as the **nested monitor call problem** [104, 158, 237], although – not only in our opinion [192] – it does not really deserve a new name.

Since the lack of expressive power of the monitor construct seems hard to overcome without a change of concept, several alternative concurrency coordination mechanisms have been proposed that are based on the monitor concept. Some of them are relevant for this survey since they have been re-introduced into COOLs.

Condition Variables

Condition variables have been proposed by Hoare [116]. The idea of this extension of the monitor concept is that an activity that has entered a monitor can block inside of the monitor at the condition variable. While it is blocked, another method call can proceed.

An activity can block by calling cond_var.**wait** where cond_var is the condition variable. The activity then blocks until another activity calls cond_var.**signal**.

Since the monitor's one-activity-at-a-time principle is still obeyed it must be specified what exactly will happen after a **signal** call, since conceptually at least two activities are ready to proceed. Is the activity that called cond_var.**signal** supposed to return from the monitor call? Will it be suspended until the signaled activity itself has terminated? Will the resuming activity wait until the signaling one terminates or blocks? An additional scheduling question is which of several activities that are blocked at the same condition variable is supposed to continue upon a **signal**?

Example. For the monitor we have shown that it leads to undesirable effects if proceed-criteria have to be implemented. One of the improvements of condition variables is that they support state proceed-criteria. This is shown in the following variant of the running example.

The following code fragment strongly relies on the monitor's one-activity-at-a-time principle because otherwise both the tests of a condition and the signalings would need an additional mutual exclusion. Concurrent execution could be allowed if all **if** statements, the two statements that change the value of is_enabled, and the signal commands would all be enclosed in critical sections.

```
class PRINT_SERVER is
public interface:
    print_text(t:STRING):INT;
    enable;
    disable;
implementation:
    enabled, disabled : CONDITION;
    is_enabled : BOOL;
    print_text(t:STRING):INT is
        if not is_enabled
        then enable.wait end;
        -- do the printing
        -- do the accounting
    end;
    enable is
        if is_enabled
        then disabled.wait end;
        ... is_enabled := true;
            enabled.signal; ...
    end;
    disable is
        if not is_enabled
        then enabled.wait end;
        ... is_enabled := false;
            disabled.signal; ...
    end;
end PRINT_SERVER;
```

Let us discuss the method enable in some more detail. The first **if** statement checks whether the print server is already enabled. In this case the current call of enable must be an additional one. Therefore, it is delayed: it waits at the condition variable disabled until the method disable is called which signals a continuation. Similarly the enabled signal activates potentially pending invocations of disable.

Discussion.

Goal ratings	
callee-oriented	yes
expressive	state proceed-criteria
isolated	no
separable	no

In comparison with the monitor concept, the expressibility of state proceed-criteria has been gained. On the other hand, the goals of isolated and separable coordination code no longer hold. This is for the same reason that has been encountered before: concurrency coordination code is mixed into the methods that implement the functionality of the class. If we would derive a subclass PRINT_SERVER_SUB with the additional methods disable_printer and enable_printer we would observe the same problems that we have discussed before for semaphores.

Moreover, it is significantly more difficult to reason about the correctness of the class implementation. In the case of the monitor, the definition of local correctness could be applied, and each method could be analized in isolation. For condition variables, sets of methods must be studied at once: the effects of all potential interleavings that might occur due to a delay of a method at a condition variable and the continuation of other methods must be taken into account. This is difficult because the conditional statements that are used to block activities and to signal their continuation are spread over the class implementation. Similar to the problem that accesses to shared data and critical code sections were spread across the whole code when programming with semaphores or when using conditional critical regions, condition variables have the same effect: If the class implementation is extended or methods must be changed, then the programmer must understand the complete coding of the class at all source code positions where the condition variable is used. Sloppy use of condition variables can easily lead to activities that are never continued, i.e. that block or deadlock for ever.

To apply the definition of local correctness for condition variables, the programmer must make sure that the class invariant holds whenever a method waits at a condition variable, because another method is continued at that time. In addition, the run-time system implementation must make sure that a signaled method only continues when the class invariant holds. With this provision, the programmer can conceptually break his methods into a set of segments, each of which can be analyzed with respect to the implication used in the definition of local correctness.

Finally, let us remark that the COOL design must make sure that condition variables are private attributes of the class. Otherwise, i.e., if condition variables can be accessed from outside the class, an activity can get control over the coordination constraints. This would contradict the goal of callee-oriented coordination.

Conditional Wait

This variant of condition variables has been introduced by Kessels [140] to improve the conditional synchronization in monitors. In contrast to condition variables which spread the code that affects them across the monitor implementation, Kessels proposed to separate the relevant conditions syntactically in the monitor.

condition identifier **is** expression;

With this declaration of the relevant condition at the beginning of the monitor implementation, later on an activity can wait simply by calling **wait**(identifier) and relying on the run-time system to signal the continuation. However, although some of the problems are solved, it is unclear and cannot be specified which of a collection of blocking activities is continued when the condition holds. Although it is much easier to identify the relevant conditions in the code, the programmer still can fail to achieve that the conditions eventually hold, i.e., activities can block or deadlock for ever.

Example. The running example using conditional wait could look like the following code fragment. Note, that this code is based on the monitor's one-activity-at-a-time principle. This cannot easily be changed, because the evaluation of the conditions and the **wait** statements must otherwise be enclosed in critical sections if concurrent execution would be allowed. If concurrent execution is allowed and the necessary critical sections are implemented then however, the same anomalies result that we have discussed for semaphores and condition variables.

```
class PRINT_SERVER is
public interface:
     print_text(t:STRING):INT;
     enable;
     disable;
implementation:
     is_enabled : BOOL;
     condition enabled is is_enabled;
     condition disabled is not is_enabled;
     print_text(t:STRING):INT is
          if not is_enabled
          then wait(enabled) end;
          -- do the printing
          -- do the accounting
          return error_code;
     end;
     enable is
          if is_enabled
          then wait(disabled) end;
          ... is_enabled := true; ...
     end;
     disable is
          if not is_enabled
          then wait(enabled) end;
          ... is_enabled := false; ...
     end;
end PRINT_SERVER;
```

Discussion.

Goal ratings	
callee-oriented	yes
expressive	state proceed-criteria
isolated	no
separable	no

Conditional Wait fulfills the goal of callee-oriented coordination code, if the condition is a private part of the class definition, i.e., if the implementation can only use conditions that are declared inside 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 and the coordination would no longer be callee-oriented.

There is no special support for expressing history proceed-criteria. Moreover, the conditional wait mechanism requires the one-activity-at-a time principle since otherwise it would be unspecified how harmful interaction of condition evaluation and method execution could be avoided.

The intermixed coordination code causes the same inheritance problems that we have discussed for semaphores: When a subclass is derived by inheritance, it is often necessary to almost duplicate existing methods, only because some of the coordination code lines must be changed. Later changes in the implementation result in duplicated code maintenance efforts along the lines of the inheritance hierarchy.

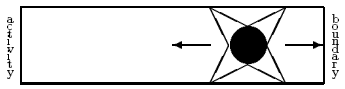
Moreover, since **wait** commands can occur at several places in the code, it is more difficult to reason about the correctness of the code. Similar to conditional waits, the definition of local correctness can be used as long as the run-time system continues methods only when the class invariant holds.

COOLs in this Category

Language	Comments
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, Conditional Wait
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

A special form of condition variable is offered by a COOL if a method can defer its own execution. In terms of condition variables, deferring is equivalent to 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. Whereas for condition variables an explicit signal must have been received by the run-time system indicating that the method can be continued, such a signal is not necessary if the method defers.

3.4.2 Handshake Control

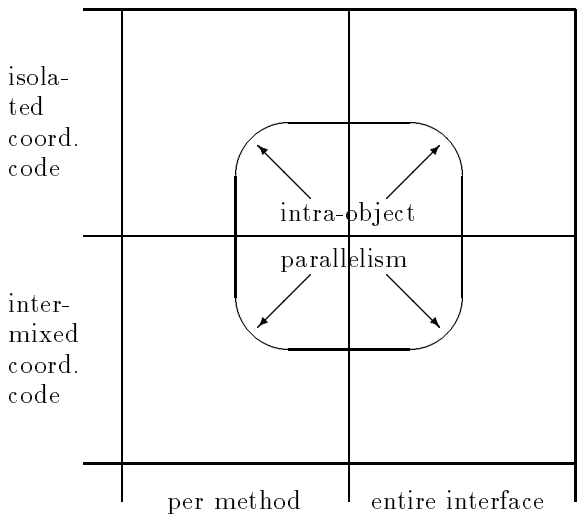


COOLs based on boundary coordination with handshake control divide the responsibility for proper coordination between the object's implementation and the run-time system. In general there is code in the class implementation that has the sole purpose of specifying the concurrency coordination, i.e, the dynamic interface of the object. The run-time system reacts according to this specification.

To reflect the fact that the class implementation influences the boundary coordination, the star has a dark circle.

In general, handshake control mechanisms fulfill the goal of callee-oriented coordination. The mechanisms are different with respect to the other goals.

Since several very different approaches to handshake control have been presented, we further refine the group of concurrency coordination mechanisms based on handshake control to structure the presentation of the mechanisms.

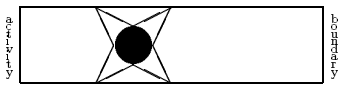


The above diagram has the following axes:

- **Isolated Coordination Code.** The upper and lower half of the diagram reflect the position of coordination code in the class implementation. The upper half is reserved for COOLs if their coordination mechanisms are implemented as individual parts of the class definition. The coordination code is isolated from code that implements the functionality. Mechanisms in this part of the diagram fulfill the goal of isolated coordination code.

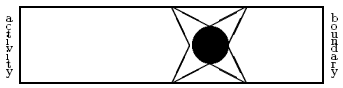
  In the pictograms handshake control with intermixed coordination code will result in the star shifted to the right:

  or, if the COOL offers additional activity centered coordination mechanisms:

- **Intermixed Coordination Code.** COOLs where the coordination mechanisms are implemented in the body of the methods that are in the public interface of the class, i.e., that implement the intended functionality fall into the lower half of the diagram. Since the coordination code is not isolated, mechanisms in this group do not fulfill the goal of isolated coordination code.

  In the pictograms handshake control with intermixed coordination code will result in the star shifted to the left:

  or, if the COOL offers additional activity centered coordination mechanisms:

- **Intra-Object-Parallelism.** A second binary dimension distinguishes COOLs that are based on the one-activity-at-a-time principle from COOLs that allow concurrent activities to execute methods at the same time. Languages and their coordination mechanisms that allow only one activity at a time are depicted in the outer part of the diagram. The COOLs with intra-object parallelism and with coordination mechanisms that allow to specify this are shown inside of the oval.

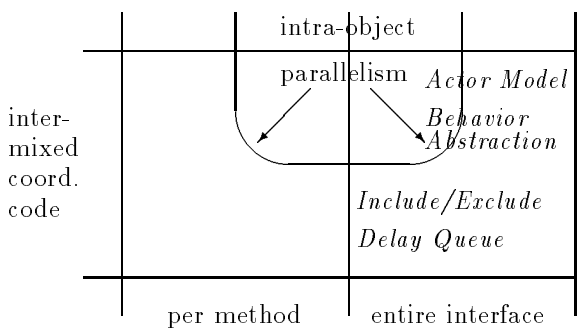
- **Per Method.** On the left hand side of the above diagram we discuss COOLs and their coordination mechanisms that affect the object's dynamic interface with respect to a single method. For example, consider a situation where a method call comes in. But before the method is executed, proceed-criteria are checked that are connected solely to this method.

- **Entire Object.** On the right hand side of the diagram are COOLs and their coordination mechanisms that affect the object's dynamic interface. For example, a method can be disabled. The code that achieves this disabling is in no direct and close relation to the affected method.

The subsequent discussion of handshake control mechanisms follows the Isolated/Intermixed dimension because this dimension has a strong impact on the interaction of concurrency and inheritance.

3.4.3 Intermixed Handshake Control

The following diagram names the mechanisms that are discussed in this section. These mechanisms fulfill the goal of callee-oriented coordination, but fall short to fulfill the goal of isolated coordination code. This causes the inheritance anomaly. Most of these mechanisms are based on the one-activity-at-a-time principle; some allow or can be extended to allow post-processing. Since the coordination code is not isolated, the goal of separable coordination code cannot be met.



All mechanisms in this group affect the dynamic interface of an object. The proceed-criteria of a method are not tightly connected to that method. Instead in general, the proceed-criteria of a method are computed in other methods.

Delay Queue

COOLs that offer delay queues conceptually allow the programmer to declare special objects of a delay queue class. This class has two methods in its public interface, namely open and close. The programmer can link delay queue objects to methods that are to be hidden from the public interface of a class. The delay queue then works as proceed-criterion. If the delay queue is open, a call of the method can proceed, otherwise it is delayed.

The programmer can implement arbitrary proceed-criteria by conditionally opening and closing the delay queues.

```
dq : DELAYQUEUE;
print_text(t:STRING):INT link dq is ...
end;
```

In the above example method print_text is linked to the delay queue dq. If dq is closed calls of print_text are delayed until the delay queues is opened by a different activity that executes a PRINT_SERVER method.[8]

Note, that the programmer can not only declare arbitrary many delay queues but that he can link several methods to the same delay queue.

Example. Below we show the running example using delay queues. To implement the intended protocol of the print server we need two delay queues (is_enabled and is_disabled). The methods print_text and disable are linked to the first delay queue. These routines can only be called after this delay queue is opened in the method enable.

```
class PRINT_SERVER is
public interface:
     print_text(t:STRING): INT;
     enable;
     disable;
implementation:
     is_enabled : DELAYQUEUE;
     is_disabled : DELAYQUEUE;
     print_text(t:STRING):INT
     link is_enabled is
          ... print and account ...
     end;
     enable link is_disabled is
          ... is_enabled.open;
              is_disabled.close...
     end;
     disable link is_enabled is
          ... is_disabled.open;
              is_enabled.close ...
     end;
end PRINT_SERVER;
```

Discussion.
Delay conditions fulfill the goal of callee-oriented coordination, provided that delay queues are private attributes of the class that is accessed concurrently.

[8] Although the link syntax is not used in any of the COOLs that offer delay queues, we think that this form is as adequate for a discussion of the underlying concept.

Goal ratings	
callee-oriented	yes
expressive	state proceed-criteria
isolated	no
separable	no

Since the whole expressive power of the COOL can be used to formulate conditions that result in opening or closing of delay queues, this mechanism certainly offers state proceed-criteria. There is, however, no support for history-proceed criteria. Moreover, the mechanism is based on the one-activity-at-a-time principle, which eases correctness considerations but limits parallel performance. It is not desirable to waive this restriction.

To see this, consider the reader-writer example. In an hypothetical implementation based on delay queues with intra-object parallelism, the conditions that allow readers or writers must be encoded explicitly, for example with counters. However, these counters must be changed inside of critical sections to make sure that they correctly reflect the current situation of executing methods. Therefore, delay queues with intra-object parallelism require an additional concurrency coordination mechanism. This has two disadvantages. First, the language gets more complicated since two different concurrency coordination mechanisms are provided, and second, the additional mechanisms will most likely be activity centered, and hence introduce the disadvantages we have discussed in section 3.3.

Again the difficulty with inheritance is caused by mixing coordination code with code that implements the intended functionality of the object. The coordination code is not isolated. Coordination code is not separable because a particular delay queue of a method can be changed in all other method bodies.

If for the running example the derived subclass is implemented which offers the new routines enable_printer and disable_printer, at least one new delay queue is required. The additional concurrency coordination constraints require that this delay queue must be opened and closed in existing methods. Therefore, the subclass implementation must almost duplicate existing methods with the only change that the new delay queue operations are added. This code duplication sacrifices the advantages of inheritance.

Include/Exclude

COOLs that offer Include/Exclude conceptually allow the programmer to include and exclude methods to and from the dynamic interface of an object.

Include/Exclude implement proceed-criteria: when a method is called that is currently excluded from the dynamic interface, the call blocks, until the method is included into the interface again.

In contrast to explicit delay queues, this mechanisms assumes the existence of a single queue of incoming calls that is "filtered" before being used to invoke methods. The include/exclude mechanisms tends to be slightly more verbose than the delay queue mechanisms since there is no way to combine several methods with similar coordination constraints into one delay queue. Instead all these methods must be included and excluded individually.

Example. The following code fragment shows the running example based on include/exclude.

Initially, only enable can be called. In the public interface section of the code this is indicated by the additional key word **initial**. After the print server has been enabled, both disable and print_text can be called whereas enable can no longer be used. Similarly, a call of disable changes the dynamic interface of the object. The dynamic interface is left unchanged by print_text.

```
class PRINT_SERVER is
public interface:
     print_text(t:STRING):INT;
     initial enable;
     disable;
implementation:
     print_text(t:STRING):INT is
          -- do the printing
          -- do the accounting
     end;
     enable is
          ...
          include {print_text, disable};
          exclude {enable};
     end;
     disable is
          ...
          exclude {print_text, disable};
          include {enable};
     end;
end PRINT_SERVER;
```

Although coordination code is somewhat separated from the functionality code there is still coordination code mixed into the functionality code, namely the **include** and **exclude** statements.

Discussion.

Goal ratings	
callee-oriented	yes
expressive	state proceed-criteria
isolated	no
separable	no

It is obvious that the include/exclude mechanism fulfills the goal of callee-oriented coordination. Since the whole expressive power of the COOL can be used to formulate conditions that lead to include or exclude operations, state proceed-criteria are fully supported. On the other hand, there is no support for history proceed-criteria.

The include/exclude mechanism is based on the one-activity-at-a-time principle which eases correctness considerations. The same arguments that have been used to explain the necessity of the one-activity-at-a-time principle of delay queues can be applied to the include/exclude mechanisms as well.

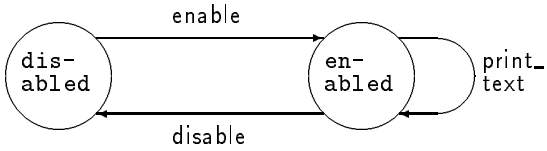
When a new subclass is derived well known inheritance problems appear because of the failure to fulfill the goal of isolated coordination code: Old methods that include/exclude certain methods into/from the dynamically available interface might want to include/exclude new routines as well. Therefore, the subclass must have almost duplications of the same methods with the sole difference that additional methods are included/excluded from the dynamic interface.

Behavior Abstractions

COOLs that offer behavior abstractions conceptually allow the programmer to dynamically change the interface of an object. Whereas 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 an automaton. There are several states; in each of these states several methods can be called. After a method is called, the object switches to another state. These states can be used to express history proceed-criteria.

In the print server example, two states `disabled` and `enabled` can be identified. In the diagram below the states are represented by circles. The arrows mark acceptable methods. Whereas invocations of enable and disable toggle between the two states, print_text does not change the state. Moreover, it can be seen that print_text can only be invoked, if the print server is in state `enabled`.



Example. The following code fragment shows the running example based on behavior abstractions.

The print server object can have two states, namely enabled and disabled. The **behavior** section of the class implementation specifies which methods are allowed to be called in each of these states.

```
class PRINT_SERVER is
public interface:
     print_text(t:STRING):INT;
     enable;
     disable;
behavior:
     enabled = {print_text, disable};
     disabled = {enable};
     initially disabled;
implementation:
     print_text(t:STRING):INT is
          -- do the printing
          -- do the accounting
          become enabled;
     end;
     enable is
          ... become enabled; ...
     end;
     disable is
          ... become disabled; ...
     end;
end PRINT_SERVER;
```

Coordination code is somewhat more isolated from the functionality code, however, there is still coordination code mixed into the functionality code, namely the **become** statements that set the next state in each of the three methods.

Discussion.

Goal ratings	
callee-oriented	yes
expressive	state proceed-criteria, history proceed-criteria
isolated	no
separable	no

Concurrency coordination mechanisms based on behavior abstractions fulfill the goal of callee-oriented co-

ordination. Moreover, as has been discussed above, in addition to state proceed-criteria the notion of states eases the expressibility of history proceed-conditions.

Whereas delay queue operations and include/exclude operations are spread across the class implementation and make it difficult to reason about which methods are hidden at any given time, behavior abstractions have the advantage of making the set of available methods visible after a method has been executed.

Behavior abstractions are based on the one-activity-at-a-time principle for the same reasons that have been 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 (or code-segment) in the implementation of a method that determines the new state, one can imagine that a concurrent method can be started immediately afterwards, even if the first method has not terminated yet. This is an application of post-processing. The increased parallelism sacrifices the goal of callee-oriented coordination unless the post-processing part guarantees that the class invariant is fulfilled at any time during its execution. If however, the class invariant is guaranteed to hold in the post-processing part, then the definition of local correctness and implication (1) can be applied without alterations.

A typical method of a class implementation based on behavior abstractions implements the functionality first. Then there is a analysis phase in which the internal state of the object is studied. Based in this analysis, the new state is determined. This leads to inheritance anomaly problems when a new subclass is derived: In the worst case new routines must add a complete case analysis to switch to the appropriate successor state. Analogously, if the set of possible states is changed there are otherwise unaffected methods that must reflect this change in their case cascade. Again, unaffected code is duplicated only because the coordination code in this method must be changed slightly.

Actor Model

The pure Actor model [6] combines the concept of objects with the concept of dynamic interfaces: If the programmer wants to change the dynamic interface of an object (called actor), this actor becomes a new actor with a different behavior. In object-oriented terminology, the new actor can even be implemented by a different class. Instead of dynamically changing whether a method can be called or not, in the Actor model the code that implements the actor behavior can be switched dynamically.

Example. We have seen in the implementation of the running example based on behavior abstractions, that there are two different behaviors (states) for the print server. In languages based on the Actor model, each of these behaviors is implemented separately. At execution time, the actor is either implemented by the code for DISABLED_PRINT_SERVER or by the code for ENABLED_PRINT_SERVER.

```
Actor DISABLED_PRINT_SERVER is
public interface:
     enable;
implementation:
     enable
         become ENABLED_PRINT_SERVER;
     end;
end DISABLED_PRINT_SERVER;

Actor ENABLED_PRINT_SERVER is
public interface:
     print_text(t:STRING): INT;
     disable;
implementation:
     print_text(t:STRING):INT
          ... print and account ...
          become ENABLED_PRINT_SERVER;
     end;
     disable:INT
          ...
          become DISABLED_PRINT_SERVER;
     end;
end ENABLED_PRINT_SERVER;
```

From the object-oriented point of view, pure Actor mechanisms have some drawbacks:

- First of all, it is in general difficult to use implementation inheritance in pure Actor languages. The reason for this is twofold: One problem is that the behavior of the actors is spread across several code segments. To derive a subclass, i.e., an actor with a specialized behavior, subclasses for each individual behavior must be defined. The other problem is that an actor can become a new actor with a totally different behavior. Because of these reasons, many Actor languages do not offer implementation inheritance at all but rely on delegation instead.
- In the above example, it is not at all obvious what is going to happen if the method

print_text is called, when the actor is in state DISABLED_PRINT_SERVER. Since the implementation neither provides a method with this name nor offers any indication that such a method will become available in a future state of the actor, many pure Actor languages do not delay the call.

Discussion.

Goal ratings	
callee-oriented	yes
expressive	state proceed-criteria, history proceed-criteria, restricted intra-object parallelism
isolated	no
separable	no

The Actor model fulfills the goal of callee-oriented coordination.

Several proposals have been made to neatly integrate the Actor model into object-oriented languages. If the code that implements different behaviors of an actor is kept together in a single class implementation, the current behavior must be expressed differently. Two general solutions can be identified. One solution is similar to behavior abstractions: the class implementation is enhanced with the notion of "state"; instead of dynamically changing the code that implements the actor behavior, actor objects change their state. The other general solution is to generate a separate object that handles the behavior issues. Depending on a current behavior, methods of the actor object are called or the calls are delayed. We consider COOLs that are based on the latter approach to be based on reflective control. They are discussed in section 3.4.5.

The pure Actor model is based on the one-activity-at-a-time principle. Whereas behavior abstractions did allow post-processing as an extension under certain conditions, the Actor model is even defined with post-processing: immediately after a **become** statement, a new method can be started while the first method can continue. To apply the definition of local correctness the post-processing part must guarantee to fulfill the class invariant at all times.

Even with post-processing the intra-object parallelism is still restricted. Another extension are **unserialized methods**. An actor has an unserialized method if that the method will not change the behavior of the actor, i.e., the **become** statement will set the current state again. In the example, the method print_text could be an unserialized method. Unserialized methods can be executed by several callers concurrently. Since the mode of a method is declared isolated from the method code, we discuss the issue of serialized/unserialized methods in more detail in section 3.4.4.

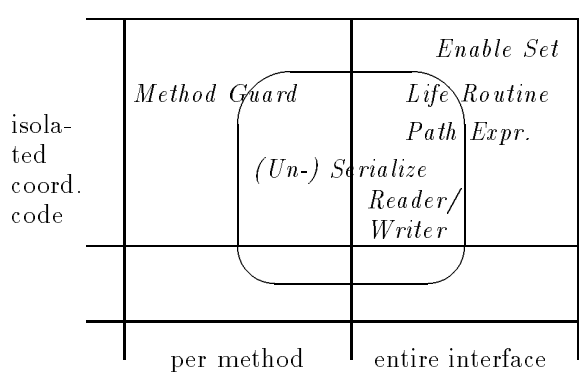
Since the coordination code is mixed into the methods that implement the intended functionality, i.e., since the goal of isolated coordination code is not met, the inheritance anomaly can be noticed. The effects are the same as for delay queues, include/exclude, and behavior abstractions.

COOLs in this Category

Language	Comments
ABCL/1	Actor
ABCL/f	Actor
Acore	Actor
ACT++	behavior abstraction
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 (allow)
Ubik	Actor

3.4.4 Isolated Handshake Control

The following diagram names the mechanisms that are discussed in this section. The mechanisms fulfill the goal of callee-oriented coordination code and are much closer in fulfilling the goal of isolated coordination code than the mechanisms in the previous section. Some mechanisms meet the goal of separable coordination code; some allow intra-object parallelism. Unfortunately, none of these mechanisms is perfect, each one still has some restrictions.



Note, that three mechanisms are located both inside and outside of the oval since these mechanisms – although originally designed for the one-activity-at-a-time principle – have straightforward extensions to handle intra-object parallelism.

Method Guard

Method guards are a way to express proceed-criteria that is similar to pre-conditions. Before a method that is guarded by a method guard is executed, the attached condition is evaluated. If the condition is evaluated to true, the method is invoked, otherwise the call is delayed.

Example. Method guards are used in the following implementation of the running example. There is a special **guard** section in the class implementation, which specifies a proceed-criterion for all methods. The methods print_text and disable can only be executed when the print server is enabled, i.e, when the private attribute state is equal to "enabled". Otherwise, only the method enable can be executed.

```
class PRINT_SERVER is
public interface:
     print_text(t:STRING):INT;
     enable;
     disable;
guards:
     print_text: state == "enabled";
     enable : state == "disabled";
     disable : state == "enabled";
implementation:
     state : "enabled", "disabled"; -- enum
     print_text(t:STRING):INT is
          -- do the printing
          -- do the accounting
     end;
     enable is
          ... state := "enabled"; ...
     end;
     disable is
          ... state := "disabled"; ...
     end;
end PRINT_SERVER;
```

The above implementation has two characteristics that should be noted. First, it relies on the one-activity-at-a-time principle. Second, the private attribute state that is used in the method guards, is modified in the routines enable and disable.

Discussion.

Goal ratings	
callee-oriented	yes
expressive	state proceed-criteria, sometimes intra-object parallelism
isolated	almost yes
separable	yes

Method guards fulfill the goal of callee-oriented coordination. Since all language features can be used to express the guarding conditions, state proceed-criteria are offered. The conditions can be arbitrary boolean expressions that use instance variables of the object. There is however no support for history proceed-conditions.

The method guards that are offered in COOLs differ with respect to the offered expressibility of intra-object parallelism. Some COOLs are based on the one-activity-at-a-time principle. Others allow to express whether a method can be executed with respect to concurrently executing (other) methods. To express these concurrency conditions, two different types of proposals have been made:

Counters. Counters are predefined functions that can 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, the number of completed method executions, and so on. Based on these counters the programmer can easily check in the guard expression whether concurrency conditions hold. Different counters are offered by different COOLs. The implementation makes sure that the counters are modified automatically.

Compatibilities. The COOL CEiffel allows to express method compatibilities, i.e., the programmer can specify the names of methods in a method guard, which 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 new call is delayed. Otherwise the new call can proceed, provided that an additional proceed-criterion is as well evaluated to true.

Although at first glance the coordination code is isolated from the code that implements functionality, there are still interdependences. Since the guards use instance variables to check the availability of a method, the coordination code is connected to the

functionality code if that uses the same instance variables. If the instance variables are changed, the outcome of the conditions might change as well. Isolation of coordination code is much better than in the mechanisms discussed in the previous sections since methods and guards can be inherited separately.

If there are interdependences between guards and instance variables then a mild form of inheritance anomaly still occurs: When a subclass is derived that adds new routines and instance variables, it is quite likely that the guards of existing methods must be adapted. For example, the new routines enable_printer and disable_printer will result in changing guards for the disable routine. In contrast to earlier approaches, the methods often remain unaffected and are not duplicated.

If intra-object parallelism is allowed, the language must define how concurrent method execution and the evaluation of proceed-criteria interact without race-conditions. Most COOLs we have looked at do not deal with this problem. Two reasons can be envisioned: either there is a hole in the language designs or the authors omitted the discussion of that problem in their presentations. The COOL SOS is an exception: in SOS there are two types of instance variables, only one type can be used in guard expressions. Therefore, in SOS coordination code is truly isolated.

Enable Set

One of the remaining problems with behavior abstractions was that each method had to perform a possibly complex analysis to determine the new behavior of the object. 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 name of the new state. The complex analysis is hidden in this method. When the sets of potential states change, hopefully only this method is affected.

Example. The following representation of the running example with Enable Sets is similar in spirit to the one that was based on behavior abstractions.

In the code the private methods enabled and disabled return Enable Sets. We call these methods state determining methods. The case analysis that determines the successor state therefore is moved into special methods and is clearly isolated from the other code.

```
class PRINT_SERVER is
public interface:
       print_text(t:STRING):INT;
       enable;
       disable;
implementation:
     private enabled:EnableSet is
       return new EnableSet(print_text,disable);
     end;
     private disabled:EnableSet is
       return new EnableSet(enable);
     end;

     print_text(t:STRING):INT is
          -- do the printing
          -- do the accounting
          become enabled;
     end;
     enable is
          ... become enabled; ...
     end;
     disable is
          ... become disabled; ...
     end;
end PRINT_SERVER;
```

The **become** statements in the methods print_text, enable, and disable call the private methods to determine the successor state.

Discussion.

Goal ratings	
callee-oriented	yes
expressive	state proceed-criteria, history proceed-criteria
isolated	almost yes
separable	yes

Enable Sets fulfill the goal of callee-oriented coordination. They support both state proceed-criteria and history proceed-criteria, in the same way as behavior abstractions do.

Enable Sets are not made for intra-object parallelism, and this restriction cannot be dropped. The reason is the same as for delay queues, include/exclude, and behavior abstractions: if intra-object parallelism would be available, this would require an additional (activity based) coordination mechanisms to guard concurrent access to instance variables that are needed to encode compatibility rules.

Compared with behavior abstractions the Enable Set mechanisms almost fulfills the goal of isolated coordination code. The reason is that code checking the internal state to determine the successor state is moved from the methods that implement class functionality to separate methods. At this new position, the coordination code can be inherited and modified separately, most often without affecting the methods that implement class functionality. Some problems remain if the state determining method uses class attributes that are used by the 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 in general possible to achieve a one-to-one-mapping between (regular) methods and private state determining methods, the coordination code is separable. In most cases it is possible to change only very few of the state determining methods when the coordination constraints must be altered.

Extension. An extension of Enable Sets has been proposed that further reduces the amount of changing code. Since at the end of each method the new state must be determined, one could envision a default method that automatically is called for this purpose. This extension removes the **become** statement from the language. If the programmer decides that a particular method needs a different algorithm to determine the successor behavior, there are syntactic means to link the additional transition routine to the method that implements the functionality.

With respect to inheritance, this extension seems to perform better than classic Enable Sets.

Path Expression

Facing the problem that growing monitor implementations lead both to sequential bottlenecks and to code for conditional synchronization that is scattered throughout the class, path expressions have been invented [50]. The idea of path expressions is to express all dependences between potentially concurrent operations at one place in each class. With a construct like

path path_list **end**

the programmer can specify, which monitor operations can be called in which order, and which operations can be executed concurrently. The path list is essentially a list of method names, enhanced in a regular expression style, i.e., choice (|), repetition ({}), concurrency (; and *) etc. can be expressed. No additional synchronization code is needed inside of the class implementation.

Example. In the following representation of the running example with path expressions several activities are allowed to invoke print_text at a time.

```
class PRINT_SERVER is
public interface:
     print_text(t:STRING):INT;
     enable;
     disable;
path:
     {enable, (print_text* | disable)};
implementation:
     print_text(t:STRING):INT is
          -- do the printing
          -- do the accounting
     end;
     enable is ... end;
     disable is ... end;
end PRINT_SERVER;
```

The **path** section of the class definition specifies in which order methods can be called. Moreover, the **path** expression defines, that several invocations of print_text can be executed concurrently.

Discussion.

Goal ratings	
callee-oriented	yes
expressive	history proceed-criteria, intra-object parallelism
isolated	yes
separable	no

Path expressions fulfill the goal of callee-oriented synchronization. The coordination code is isolated and can be inherited separately.

Although path expressions are elegant for expressing concurrency constraints inside of the class, they inherit the monitor's deficiency to properly express conditional synchronization [36]: it cannot be expressed, if an method can only be executed if the object's state fulfills certain conditions. Similarly, it is unclear which of several invocations of a particular method will be activated when the path expression allows one of them to proceed.

Consider for example a derived subclass that allows only a certain number of concurrently executing jobs. Since this new condition cannot be expressed with path expressions an implementation must make state

checking methods available to the outside and must augment the print_text method with an additional error code to indicate an illegal call. Therefore, conditional behavior is implemented by the caller, which breaks modularity.

Some COOLs offer concurrency operators (; and *) and hence intra-object parallelism. The concurrency coordination mechanisms is perfectly capable of expressing this. However, there are still some other COOLs that are based on the one-activity-at-a-time principle.

The main inheritance problem of path expressions is their lacking separability. A subclass can either inherit the whole path expression of its ancestor, or completely redefine it. There is no way to just alter a part of a path expression. The redefinition of complex path expressions requires an in-depth understanding of all potential concurrent constellations, and is thus not easy. We consider this lack to induce a mild form of inheritance anomaly.

Life Routine

Let us now discuss several proposed language constructs that can be used in life routines. All these constructs use the "message" terminology instead of understanding the message as a method call.

In general, life routines process one incoming call at a time. This strictly serializes the object's behavior. One way towards intra-object parallelism is to offer an additional construct to initiate concurrency. For example, if the COOL offers a **fork** command, the life routine can use this command to execute the requested method concurrently. Above that, the life routine can do the book-keeping to check whether an incompatible method is currently processed.

Receive Statement. The **receive** statement can be used to explicitly wait for the arrival of a method. In several languages, this statement is enhanced with a condition that must hold when a message is extracted (**receive** ... **when**). A special form of this condition is to wait for a message from a particular sender (**receive** ... **from**).

Guarded Commands. Another well known proposal for coordination in life routines, called **guarded commands**, was introduced by Dijkstra [89]:

if $G_1 \longrightarrow \mathsf{StmtList}_1$
[] $G_2 \longrightarrow \mathsf{StmtList}_2$
...
[] $G_n \longrightarrow \mathsf{StmtList}_n$
end

If in the above **if** statement one of the boolean conditions (guards) G_1–G_n holds, the corresponding list of statements is executed, for example a particular message can be received. If several guards hold, then one of them is selected randomly. Because of their popularity, we stress the fact the guarded commands are often provided in the syntactic form of **select** statements.

Example. We use the implementation of the print server as shown in section 2.5 and refine it here. Inside the **select** statement there are two guards that check the same condition, namely is_enabled. If is_enabled is true, the life routine accepts incoming calls of either print_text or disable.

```
class PRINT_SERVER is
public interface:
     print_text(t:STRING):INT;
     enable;
     disable;
implementation:
     is_enabled:BOOL;
     life body is
          loop -- forever
               select
               [] is_enabled
                  -> receive "print_text(t)";
                     fork my_print_text(t);
               [] not is_enabled
                  -> receive "enable";
                     my_enable;
               [] is_enabled
                  -> receive "disable";
                     my_disable;
               end;
          end;
     end;
     -- these routines cannot be called directly
     my_print_text(t:STRING):INT is ... end;
     my_enable is ... end;
     my_disable is ... end;
end PRINT_SERVER;
```

The above code allows for intra-object parallelism: if the printer is enabled and if print_text is called, the **select** statement enters its first branch. Inside this

branch the method my_print_text is started with a **fork** statement. Therefore, the life routine is immediately able to accept the next incoming method. Calls of enable and disable are not spawned for concurrent execution; while **my_enable** and my_disable are processed, further method calls will be delayed.

Discussion.

Goal ratings	
callee-oriented	yes
expressive	state proceed-criteria, intra-object parallelism possible
isolated	yes
separable	no

Life routines fulfill the goal of callee-oriented coordination code.

Coordination code is isolated in the life routine, provided that the programmer manages to put only coordination code into the life routine and is not tempted to code class functionality in the life routine. With life routines it depends on the programming style that no algorithmic details (i.e. parts of the functionality of the object) are implemented in the life routine. Indicators for an unwanted mix of functionality code and coordination code in classes with life routines are **receive** statements that are spread across the whole class code or if the StmtLists of guarded commands contain significantly more code than an invocation of a private method that implements intended object behavior.

Similar to path expressions, the life routine can only be inherited as a whole, i.e., the coordination code is not separable. In most cases the programmer will need to reprogram the complete life routine in subclasses. Below we discuss an extension of life routines that addresses the inheritance problem, i.e., that allows to re-use standardized life routines.

Inside of life routines, all language features can be used to encode state proceed-criteria. However, there is no special support for history proceed-criteria. Although life routines alone do not offer intra-object parallelism this can easily be added, e.g., with a **fork** statement. The language designer then again faces the trade-off between easy correctness considerations and increased performance.

Generalized Life Routine. In the COOL Eiffel// standard life routines can be inherited from a library of generalized life routines. The programmer provides both the methods that should be executed and implements 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 pre-requisite of inherited life routines is that methods are first class citizens of the language. Without this property, there is no way in which the programmer could talk about guard functions and methods and use them as entries in the table that is handled by the life routine.

If such generalized life routines are used, inheritance works fine. In fact, 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 external control, where the coordination constraints are external and thus do not interfere with inheritance.

Another advantage of generalized life routines is that the potential for an undesirable mix of coordination and functionality code can be restricted, i.e., the goal of isolated coordination code is met directly and without the need to rely on a particular programming style.

(Un-)Serialized Method and Object

Several COOLs which otherwise rely on the one-activity-at-a-time principle allow to label methods or even objects as "unserialized". Unserialized objects are guaranteed not to store any internal state. All unserialized methods are purely functional and free of side-effects. Similarly, unserialized methods, can be executed by several concurrent activities.

Although this labeling technique is a step towards the expressibility of intra-object parallelism, there is no way to express for example the mutual incompatibility of two methods, each of which can be executed concurrently in absence of an execution of the other method.

COOL designers that offer (un-)serialized methods or objects often feel a need for an additional activity centered coordination mechanism to coordinate concurrently executing methods which are "almost" pure functional. The language feature table below gives more details by naming the additional coordination mechanism where appropriate. If no additional coordination mechanism is provided, the COOL relies on the fact that the programmer uses unserialized methods only when they are indeed pure functional.

Discussion.

Goal ratings	
callee-oriented	yes
expressive	restricted intra-object parallelism
isolated	yes
separable	yes

(Un-)serialized methods fulfill most goals, except that they offer very limited expressive power. Because of this restriction this construct is often combined with other mechanism to gain the desired expressive power. In such combinations, the other mechanism determines whether and which goals are met.

Reader/Writer-Protocol

Another approach to enhance the expressibility of concurrency conditions is offered by COOLs that allow to label methods to be readers (sometimes called observers) and writers (sometimes modifiers). If the methods of a class are labeled accordingly, the runtime system can ensure that modifying methods have exclusive access to the object whereas several observing methods can be executed concurrently.

Discussion.

Goal ratings	
callee-oriented	yes
expressive	restricted intra-object parallelism
isolated	yes
separable	yes

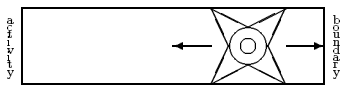
See the discussion for (un-)serialized methods.

COOLs in this Category

Language	Comments
Acore	unserialized method
ASK	serialized
Arche	enable set, reader/writer
Blaze-2	serialized method (additional: lock)
CEiffel	method compatibility, method guard; clear separation between the two types of guards
CLIX	method guard
COB	life routine
Comp. C++	serialized (additional: coordination future)
Conc. Aggregate	unserialized (additional: reader/writer lock)
Conc.Class Eiffel	life routine
cooC	serialized (additional: semaphore)
COOL (Stanford)	serialized (additional: condition variable)
Demeter	serialized, method guard
Distr. Eiffel	method guard, reader/writer
Distr. Smalltalk – Process	method guard, serialized (additional: semaphore)
Dragoon	method guard (counter)
Eiffel//	life routine, 1st class methods
Guide	method guard (counter)
HAL	method guard (negative)
Java	serialized (additional: mutex)
Mediators	life routine (receive), method guard (counter)
Mentat	life routine (receive)
Meyer's Proposal	method guard
Micro C++	life routine, (receive)
Obliq	serialized (additional: mutex, lock)
Orca	method guard
Parallel Computing Action	method guard (negative)
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)

Some COOLs offer two coordination mechanisms. For example, Procol offers both path expressions and method guards. Especially the serialized methods and the reader/writer-protocol can easily be used in combination with some other mechanisms.

3.4.5 Reflective Control



COOLs based on boundary coordination with reflective control keep class implementations free of coordination code. In contrast to external control, where there is no explicit coordination code at all, mechanisms based on reflective control enable the programmer to explicitly formulate the coordination constraints. This can be done in meta-classes.

Example. The following code is the well known implementation of the PRINT_SERVER. In addition, there is a class PRINT_SHADOW that has two methods in its public interface, namely entry and exit.

```
class PRINT_SERVER is
public interfacee:
     print_text(t:STRING):INT;
     enable;
     disable;
implementation:
     print_text(t:STRING):INT is ... end
     enable is ... end;
     disable is ... end;
end PRINT_SERVER;

class PRINT_SHADOW is
public interface:
     entry is ... end;
     exit is ... end;
end PRINT_SHADOW;

PRINT_SERVER::attach(ps,pshadow,entry,exit);
```

The last code line dynamically links the shadow object pshadow to the print server object ps. After this dynamic link an incoming method call for ps will first start the method entry of the shadow object, then invoke the called method of ps, and finally call the method exit of the shadow object. Therefore, the shadow object can implement any form of delay.

Discussion.

Goal ratings	
callee-oriented	yes
expressive	state proceed-criteria, intra-object parallelism
isolated	yes
separable	no, possibly yes.

Reflective control mechanisms fulfill the goals of callee-oriented and isolated coordination code. The expressibility is limited, since there is no support for history proceed-criteria.

Whether the goal of separable coordination code is fulfilled depends on the language mechanisms. If the shadow object can be used like a generalized life routine and the programmer refrains from implementing functionality in the shadowing class that belongs into the shadowed class, the coordination code can be separable. Otherwise it is clearly not: to change the coordination constraints the shadowing class must be completely re-programmed.

COOLs in this Category

Language	Comments
ABCL/R2	
ABCL/R3	
DROL	protocol object, one-activity-at-a-time
MeldC	shadow object, concurrent access possible

4 MAPPING AND LOCATION

...will be added in a later version of the survey.

5 LANGUAGE DISCUSSION

5.1 General Language Design Issues

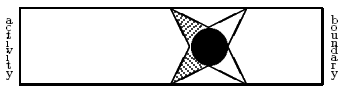
Language versus Library. Some COOLs extend existing object-oriented languages by adding concurrency in a library. Special classes are offered, that can be mixed-in to add for example semaphore operations. Buhr points out in [45] that there are reasons why such library extensions can in general not be implemented correctly. The basic idea of Buhr's argumentation is as follows. Since the compiler does not know about the special concurrent semantics of the the added libraries 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 mixed-in library class.

a.**lock**;
a.method;
a.**unlock**

Inside of the cirtical section between the call of **lock** and the call of **unlock** the above code executes a method 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 method 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 method. This "optimization" destroys the correctness of the code and make erratic behavior almost impossible to debug. Buhr points out several related problems all of which can be explained by the inawareness of a standard compiler of the additional concurrent semantics.

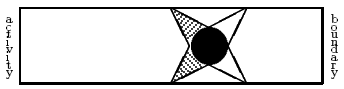
5.2 Language Survey

ABCL/1
[244]



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
Akinori Yonezawa ⟶ yonezawa@is.s.u-tokyo.ac.jp

ABCL/f
[219]



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

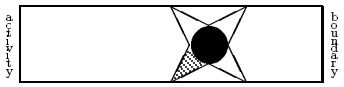
ABCL/R2
[168, 244]

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
Akinori Yonezawa ⟶ yonezawa@is.s.u-tokyo.ac.jp

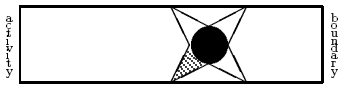
ABCL/R3
[169]

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

Acore
[166]

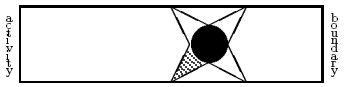


ACT++
[129, 130, 131, 132, 133]

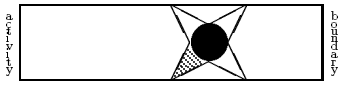


ftp://actor.cs.vt.edu/pub
Dennis Kafura ⟶ kafura@cs.vt.edu

Act1

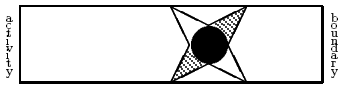


Actalk
[44]



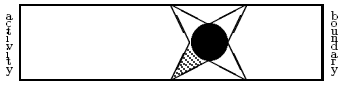
http://web.yl.is.s.u-tokyo.ac.jp/members/briot/actalk/actalk.html
ftp://camille.is.s.u-tokyo.ac.jp/pub/members/briot/actalk
ftp://ftp.ibp.fr/ibp/softs/litp/actalk
Jean-Pierre Briot ⟶ briot@is.s.u-tokyo.ac.jp

ActorSpace
[5, 49]



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
[171, 222]



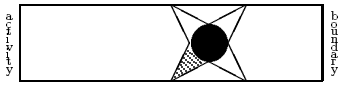
Amber
[70]

A-NETL
[90, 246]

Arche
[32]

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

ASK
[205]



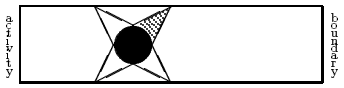
Guilia Iannello ⟶ iannello@udsab.dia.unisa.it

A'UM
[245]

BETA
[42, 164, 163]

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

Blaze 2
[175, 176]



Piyush Mehrotra ⟶ pm@icase.edu

Braid, Data-Parallel Mentat
[234, 235]

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

C**
[150, 151, 152]

James R. Larus ⟶ larus@cs.wisc.edu

Cantor
[20]

CEiffel
[159, 160]

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

CFM
[230]

CHARM++
[136]

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
[120]

COB
[118]

Compositional C++, CC++
[54, 67, 68, 91]

ftp://csvax.cs.caltech.edu/comp/CC++
K. Mani Chandy ⟶ mani@vlsi.caltech.edu
Carl Kesselman ⟶ carl@vlsi.caltech.edu

Concurrency Class for Eiffel
[138, 139]

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

Concurrent Aggregates, CA
[74] – [75], [137, 196]

http://www-csag.cs.uiuc.edu
ftp://cs.uiuc.edu/pub/csag
group ⟶ concert@red-herring@cs.uiuc.edu
Andrew A. Chien ⟶ achien@cs.uiuc.edu

ConcurrentSmalltalk
[243]

cooC
[227]

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

COOL (Chorus)
[7, 153, 154]

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

COOL (NTT), ACOOL
[167]

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

COOL (Stanford)
[64, 65, 66]

ftp://cool.stanford.edu
Rohit Chandra ⟶ rohit@cool.stanford.edu

Coral
[69]

CST, Concurrent Smalltalk (MIT)
[81, 117]

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

Demeter
[161]

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

Distributed C++, DC++
[61, 62]

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

Distributed Eiffel
[103]

Distributed Smalltalk – Object
[31, 84, 172, 181, 208]

Distributed Smalltalk – Process
[157]

DoPVM
[110]
ftp://mathcs.emory.edu/pub/vss
Contact V. S. Sunderam ⟶ vss@mathcs.emory.edu
Charles Hartley ⟶ skip@mathcs.emory.edu

DOWL, distributed Trellis/Owl
[1, 2]
Bruno Achauer ⟶ bruno@tk.telematik.informatik.uni-karlsruhe.de

dpSather
[209]
Heinz Schmidt ⟶ Heinz.Schmidt@fcit.monash.edu.au

Dragoon
[21, 22]
Colin Atkinson ⟶ atkinson@cl.uh.edu
Marco De Michele ⟶ demichel@txt.it

DROL
[216]

Eiffel//
[55, 56, 57, 58, 59, 60]
Denis Caromel ⟶ caromel@mimosa.unice.fr

Ellie
[12, 13]
ftp://ftp.diku.dk/diku/dists/ellie/papers
Birger Andersen ⟶ andersen@informatik.uni-kl.de

Emerald
[121, 127, 128]
ftp://ftp.diku.dk/pub/diku/dists/emerald
Eric Jul ⟶ eric@diku.dk

EPEE, Eiffel Parallel Execution Env.
[106, 124, 125, 126]
Jean-Marc Jézéquel ⟶ jezequel@irisa.fr

ES-Kit Software
[210, 223]
http://www.mcc.com

ESP – Extensible Software Platform
[71]
David Croley ⟶ croley@mcc.com
Arun Chatterjee ⟶ arun@mcc.com

Fleng++
[217]

Fragmented Objects, FOG/C++
[94, 165]
Yvon Gourhand ⟶ gourhant@corto.inria.fr

Guide
[73, 85, 105, 144, 146, 202]
http://www.imag.fr
ftp://ftp.imag.fr/pub/GUIDE

HAL
[119, 141]
ftp://biobio.cs.uiuc.edu/pub/Hal
Chris Houck ⟶ chouck@ncsa.uiuc.edu
Wooyoung Kim ⟶ wooyoung@cs.uiuc.edu
Gul Agha ⟶ agha@cs.uiuc.edu

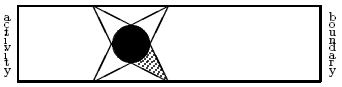
Harmony
[162]

Heraklit
[114]
http://www2.informatik.uni-erlangen.de/IMMD-II/Research/Projects/HERAKLIT
Peter Arius ⟶ arius@informatik.uni-erlangen.de
Wolfgang Betz ⟶ betz@informatik.uni-erlangen.de

HoME
[185]

Hybrid
[182, 183, 189]
Oscar Nierstrasz ⟶ oscar@iam.unibe.ch

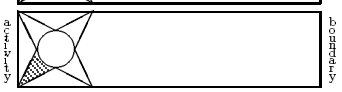
Java



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

Karos
[102]

LO
[15]



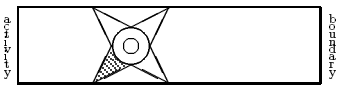
Maude

?

⟶

Mediators
[95]

MeldC
[134, 135, 197]

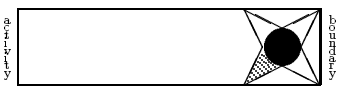


group ⟶ MeldC@cs.columbia.edu
Gail E. Kaiser ⟶ kaiser@cs.columbia.edu

Mentat
[96, 97, 98, 99, 100, 101]

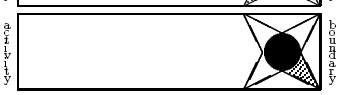
http://www.cs.virginia.edu/ mentat
ftp://uvacs.cs.virginia.edu
group ⟶ mentat@virginia.edu
Andrew S. Grimshaw ⟶ grimshaw@virginia.edu

Meyer's Proposal
[179]



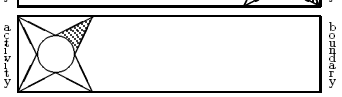
Betrand Meyer ⟶ bertrand@eiffel.com

Micro C++, μC++
[47, 46, 48]



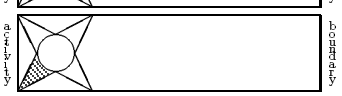
ftp://plg.uwaterloo.ca/pub/uSystem
group ⟶ usystem@maytag.uwaterloo.ca
Peter A. Buhr ⟶

Modula-3*
[111]



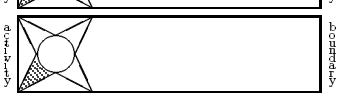
Ernst A. Heinz ⟶ heinze@ira.uka.de

MPC++
[122, 123]

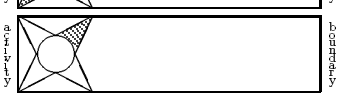


http://www.rwcp.or.jp
Yutaka Ishikawa ⟶ ishikawa@rwcp.or.jp

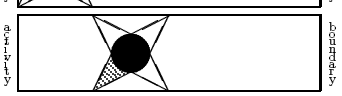
Multiprocessor Smalltalk
[187]



NAM
[155]



Obliq
[51, 52]

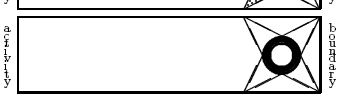


http://www.research.digital.com/SRC/home.html
Luca Cardelli ⟶ luca@src.dec.com

Orca
[24, 25, 27, 26, 112, 218]

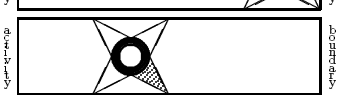
ftp://ftp.cs.vu.nl/pub/amoeba/orca_papers
ftp://ftp.cs.vu.nl/pub/papers/orca
Henri E. Bal ⟶ bal@cs.vu.nl

Oz, Perdio
[113, 211, 212, 213]



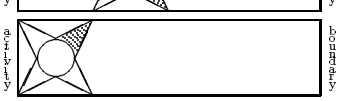
ftp://ps-ftp.dfki.uni-sb.de
http://ps-www.dfki.uni-sb.de/oz
group ⟶ oz@dfki.uni-sb.de
Gerd Smolka ⟶ smolka@dfki.uni-sb.de

Panda
[19]



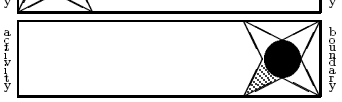
ftp://ftp.uni-kl.de/reports_uni-kl/computer_science/system_software
http://www.uni-kl.de/AG-Nehmer/panda/panda.html
Holger Assenmacher ⟶ assen@informatik.uni-kl.de
Reinhard Schwarz ⟶ schwarz@informatik.uni-kl.de

Parallel C++, pC++
[37, 38, 156]



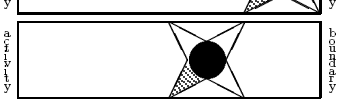
http://www.extreme.indiana.edu/sage
Dennis Gannon ⟶ gannon@cs.indiana.edu

Parallel Computing Action
[203, 204]



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

Parallel Object-Oriented Fortran
[201]



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

PO
[78, 79]

POOL, POOL-T, POOL-I
[8, 9, 10, 11, 215, 236]

Presto
[33, 34]
ftp://ftp.cs.washington.edu/pub

Procol
[41, 147, 148]
http://www.leidenuniv.nl

Proof
[242]

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

PVM++
[198]
Roland Pozo ⟶ pozo@cs.utk.edu

QPC++
[39]
Dietrich Boles ⟶ boles@informatik.uni-oldenburg.de

Rosette
[224, 225, 226]
ftp://biobio.cs.uiuc.edu

SAM
[199]

Scheduling Predicates
[173, 174]
http://www.dsg.cs.tcd.ie/research/sos.html
Alexis Donnelly ⟶ donnelly@cs.tcd.ie
Seán Baker ⟶ baker@cs.tcd.ie

Scoop
[231]

Smalltalk-80
[93]

Sos
[173]
http://www.dsg.cs.tcd.ie/research/sos.html
Alexis Donnelly ⟶ donnelly@cs.tcd.ie
Seán Baker ⟶ baker@cs.tcd.ie

Synchronizing Resources, SR
[17, 16, 186]
ftp://ftp.cs.arizona.edu//sr
http://www.cs.arizona.edu/sr/www
group ⟶ sr-project@cs.arizona.edu
Gregory R. Andrews ⟶ greg@cs.arizona.edu

Tool
[63]
http://www.inf.puc-rio.br/~sergio/tool
Sergio E. R. de Carvalho ⟶ sergio@inf.puc-rio.br

Trellis/Owl
[180, 206, 207]

Ubik
[83]
Peter De Jong ⟶ pdjong@vnet.ibm.com

UC++
[238]
http://www.cs.ucl.ac.uk/coside/ucpp
Russel Winder ⟶ R.Winder@cs.ucl.ac.uk

6 CONCLUSION

Parallel programming is difficult, at least with the programming languages that are available today. The combination of parallel and object-oriented paradigms in the design of COOLs raises various difficulties, since both paradigms have some contradictory issues.

An understanding of the tension between both concepts and the corresponding language design space is crucial for the discussion and the design of COOLs. By condensing the wide spread information about COOLs in a single survey, we hope to provide a useful map to the topology of the land of COOLs which might help to prevent further re-inventions and might guide future COOL endeavors.

Acknowledgements

Most of the research that resulted in this survey has been done during my year long stay at ICSI. Thanks to Jerome Feldman for the support and the discussions during that year. Thanks to Seth C. Goldstein for helping to find and structure the material presented in this survey. Thanks to Wolf Zimmermann and Claudio Fleiner for commenting on earlier drafts.

References

[1] Bruno Achauer. The DOWL distributed object-oriented language. *Communications of the ACM*, 36(9):48–55, September 1993.

[2] Bruno Achauer. Implementation of distributed Trellis. In *Proc. of ECOOP'93 – 7th European Conf. on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 103–117, Kaiserslautern, Germany, July 26–30, 1993. Springer-Verlag Berlin, Heidelberg, New York.

[3] W. B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, February 1982.

[4] *The Programming Language Ada Reference Manual*, 1983.

[5] Gul Agha and Christian J. Callsen. ActorSpaces: An open distributed programming paradigm. In *Proc. of the 4th ACM Symp. on Principles & Practice of Parallel Programming*, pages 23–32, May 1993. Appears also as ACM SIGPLAN Notices 28(7), July 1993.

[6] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. MIT Press Cambridge, Massachusetts, London, England, 1986.

[7] Paulo Amaral, Christian Jacquemot, Peter Jensen, Rodger Lea, and Adam Mirowski. Transparent object migration in COOL2. In *Proc. of the Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems, ECOOP'92*, Utrecht, The Netherlands, June 29, 1992.

[8] P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proc. of ECOOP'87 – European Conf. on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, pages 234–242, Paris, France, June 15–17, 1987. Springer-Verlag Berlin, Heidelberg, New York.

[9] Pierre America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press Cambridge, Massachusetts, London, England, 1987.

[10] Pierre America. A parallel object-oriented languag with inheritance and subtyping. In *Proc. of ECOOP OOPSLA'90, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 161–168, Ottawa, Canada, October 21–25, 1990.

[11] Pierre America. POOL: Design and experience. In *Proc. of the ECOOP OOPSLA Workshop on Object-Based Concurrent Programming*, pages 16–20, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[12] Birger Andersen. Ellie - a general, fine-grained, first class object based language. *Journal of Object-Oriented Programming*, 5(2):35–42, May 1992.

[13] Birger Andersen. Efficiency by type-guided compilation. In *Proc. of the Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pages e1–e5, OOPSLA'93, Washington D.C., September 27, 1993.

[14] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[15] Jean-Marc Andreoli, Remo Pareschi, and Marc Bourgois. Dynamic programming as multiagent programming. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proc. of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, pages 163–176, Geneva, Switzerland, July 15–16, 1991. Springer-Verlag Berlin, Heidelberg, New York.

[16] Gregory R. Andrews. Synchronizing resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, October 1981.

[17] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings Publishing Company, 1993.

[18] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[19] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. PANDA – supporting distributed programming in C++. In *Proc. of ECOOP'93 – 7th European Conf. on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 361–383, Kaiserslautern, Germany, July 26–30, 1993. Springer-Verlag Berlin, Heidelberg, New York.

[20] W. C. Athas and N. J. Boden. Cantor: An Actor programming system for scientific computing. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 66–68, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[21] Colin Atkinson, Stephen Goldsack, Andrea Di Maio, and Rami Bayan. Object-oriented concurrency and distribution in DRAGOON. *Journal of Object Oriented Programming*, 4(1):11–20, March/April 1991.

[22] Colin Atkinson, Andrea Di Maio, and Rami Bayan. Dragoon: an object-oriented notation supporting the reuse and distribution of ada software. In *Ada Letters*, pages 50–59, Fall 1990.

[23] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[24] Henri E. Bal. A comparative study of five parallel programming languages. In *proc. of Spring '91 Conf. on Open Distributed Systems, EurOpen*, pages 209–228, Tromso, Norway, May 20–24 1991.

[25] Henri E. Bal. Comparing data synchronization in Ada9X and Orca. Technical Report IR-345, Vrije Universiteit, Amsterdam, The Netherlands, December 1993.

[26] Henri E. Bal and M. Frans Kaashoek. Object distribution in Orca using compile-time and run-time techniques. In *Proc. of OOPSLA'93, 8th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 162–177, Washington D.C., 26 September – 1 October, 1993. ACM SIGPLAN Notices 28(10).

[27] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[28] Henry E. Bal, Jennifer S. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[29] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1988.

[30] J. P. Banning. An efficient way to find the side-effects of procedure calls and the aliases of variables. In *Proc. of the 6th ACM Symp. on Principles of Programming Languages*, pages 29–41, San Antonio, TX, January 1979.

[31] John K. Bennet. The design and implementation of distributed Smalltalk. In *Proc. of OOPSLA'87, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 318–330, Orlando, Florida, October 4–8, 1987. ACM SIGPLAN Notices 22(12).

[32] Marc Benveniste and Valérie Issarny. Concurrent programming notations in the object-oriented language Arche. Technical Report 690, IRISA, Institut de Recherche en Informatique et Systems Aleatoires, December 1992.

[33] B. N. Bershad, E. D. Lazowska, and H. M. Levy. Presto: A system for object-oriented parallel programming. *Software – Practice and Experience*, 1998.

[34] Brian. N. Bershad. The PRESTO user's manual. Technical Report 88-01-04, Department of Computer Science, University of Washington, Seattle, January 1988.

[35] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Gregory Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *3rd Annual ACM Symp. on Parallel Algorithms and Architectures*, Hilton Head, South Carolina, July 21-24, 1991.

[36] T. Bloom. Evaluating synchronization machanisms. In *Proc. of the 7th Symp. on Operating Systems Principles*, pages 24–32, Pacific Grove, CA, December 1979.

[37] François Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), 1993.

[38] François Bodin, Peter Beckman, Dennis Gannon, Shelby X. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proc. of Supercomputing'93*, pages 588–597, Portland, Oregon, November 15–19, 1993.

[39] Dietrich Boles. Parallel object-oriented programming with QPC++. *Structured Programming*, 14:158–172, 1993.

[40] A. H. Borning. Classes versus prototypes in object-oriented languages. In *Proc. of the ACM/IEEE Fall Joint Computer Conf.*, 1986.

[41] Jan van den Bos and Chris Laffra. PROCOL: A parallel object language with protocols. In *Proc. of OOPSLA'89, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 95–102, New Orleans, Louisiana, October 1–6, 1989. ACM SIGPLAN Notices 24(10).

[42] Soren Brandt and Ole Lehrmann Madsen. Object-oriented distributed programming in BETA. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proc. of the ECOOPO'93 Workshop on Object-Based Distributed Programming*, number 791 in Lecture Notes in Computer Science, pages 185–212, Kaiserslautern, Germany, July 26–27, 1993. Springer-Verlag Berlin, Heidelberg, New York.

[43] Jean-Piere Briot and Akinori Yonezawa. Inheritance and synchronization in concurrent oop. In *Proc. of ECOOP'87 – European Conf. on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, pages 33–40, Paris, France, June 15–17, 1987. Springer-Verlag Berlin, Heidelberg, New York.

[44] Jean-Pierre Briot. From objects to Actors: Study of a limited symbiosis in Smalltalk-80. Technical Report 88-58, Laboratoire Informatique Théorique et Programmation, LITP, Paris, France, September 1988.

[45] Peter A. Buhr. Are safe concurrency libraries possible? *Communications of the ACM*, 38(2):117–120, February 1995.

[46] Peter A. Buhr and Glen Ditchfield. Adding concurrency to a programming language. In *Proc. of USENIX C++ Technical Conference*, pages 207–223, Portland, OR, August 10–13, 1992.

[47] Peter A. Buhr, Glen Ditchfield, Richard A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: concurrency in the object-oriented language C++. *Software – Practice and Experience*, 22(2):137–172, February 1992.

[48] Peter A. Buhr and Richard A. Stroobosscher. *μC++ Annotated Reference Manual, Version 3.7*. University of Waterloo, June 1993.

[49] Christian J. Callsen and Gul Agha. Open heterogeneous computing in ActorSpace. *Journal of Parallel and Distributed Computing*, 21(3):289–300, June 1994.

[50] R. H. Campbell and A. N. Habermann. The specification of synchronization by path expressions. *Lecture Notes of Computer Science*, 16:89–102, 1974.

[51] Luca Cardelli. Obliq: A language with distributed scope. Technical Report 122, Digital Equipment Corporation, Systems Research Center, 1994.

[52] Luca Cardelli. A language with distributed scope. *Computing System*, 8(1):27–59, January 1995.

[53] Luca Cardelli and Peter Wegner. On understanding types, data abstractions, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[54] Peter Carlin, Mani Chandy, and Carl Kesselman. *The Compositional C++ Language Definition, Revision 0.9*. California Institute of Technology, Pasadena, March 1, 1993.

[55] Denis Caromel. A general model for concurrent and distributed object-oriented programming. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 102–104, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[56] Denis Caromel. Service, asynchrony and wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–22, November 1989.

[57] Denis Caromel. Programming abstractions for concurrent programming. In *Proc. of Conf. on Technology of Object-Oriented Languages and Systems, TOOLS Pacific'90*, pages 245–253, Sydney, Australia, November 1990.

[58] Denis Caromel. A solution to the explicit/implicit control dilemma. In *Proc. of ECOOP OOPSLA'90, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 21–25, Ottawa, Canada, October 21–25, 1990.

[59] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[60] Denis Caromel and Manuel Rebuffel. Object-based concurrency: Ten language features to achieve reuse. In *Proc. of Conf. on Technology of Object-Oriented Languages and Systems, TOOLS USA'93*, pages 205–214, Santa Barbara, CA, August 1993.

[61] H. Carr, R. R. Kessler, and M. Swanson. Distributed C++. *ACM SIGPLAN Notices*, 28(1):81, January 1993.

[62] Harold Carr, Robert R. Kessler, and Mark Swanson. Compiling distributed C++. In *Proc. 5th Symp. on Parallel and Distributed Processing*, pages 496–503. IEEE Computer Society, December 1993.

[63] Sergio E. R. de Carvalho. The object and event oriented language TOOL. Technical Report MCC06-93, Pontificia University, Rio de Janeiro, Brazil, 1993.

[64] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: A language for parallel programming. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 126–148. MIT Press Cambridge, Massachusetts, London, England, 1990.

[65] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in COOL. In *ACM Sigplan Symp. on Principles and Practice of Parallel Programming*, pages 249–259. ACM Press, New York, September 7–8, 1993.

[66] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[67] K. Mani Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. In *Proc. of the 5th Int. Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 124–144, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag Berlin, Heidelberg, New York.

[68] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 281–313. MIT Press Cambridge, Massachusetts, London, England, 1993.

[69] Daniel T. Chang. CORAL: A concurrent object-oriented system for constructing and executing sequential, parallel and distributed applications. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 26–30, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[70] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. Technical Report 89-04-01, Department of Computer Science, University of Washington, Seattle, September 1989.

[71] Arun Chatterjee. Distributed execution of C++ programs. In *Proc. of the Workshop on Efficient*

Implementation of Concurrent Object-Oriented Languages, pages b1–b6, OOPSLA'93, Washington D.C., September 27, 1993.

[72] Doreen Y. Cheng. A survey of parallel programming languages and tools. Technical Report RND-93-005, NASA, Ames Research Center, Moffett Field, CA, March 1993.

[73] P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte, and X. Rousset de Pina. Experience with shared object support in the Guide system. *Symp. on Experiences on Distributed Systems and Multiprocessors*, 93.

[74] Andrew A. Chien. Concurrent Aggregates: Using multiple-access data abstractions to manage complexity in concurrent programs. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 31–36, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[75] Andrew A. Chien, Vijay Karamcheti, John Plevyak, and Xingbim Zhang. *Concurrent Aggregates (CA) Language Report*. Concurrent Systems Architecture Group, Department of Computer Science, University of Illinois at Urbana-Champaign, February 1994.

[76] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proc. of the 20th ACM Symp. on Principles of Programming Languages*, pages 232–245, Charleston, SC, January 1993.

[77] M. E. Conway. A multiprocessor system design. In *Proc. of the AFIPS Fall Joint Computer Conf.*, pages 139–146, Las Vegas, November 1963.

[78] Antonio Corradi and Letizia Leonardi. PO an object model to express parallelism. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 152–155, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[79] Antonio Corradi, Letizia Leonardi, and Daniele Vigo. Massively parallel programming environments: How to map parallel objects on transputers. In M. Becker, L. Litzler, and M. Trehel, editors, *Proc. of Transputers '92, Advanced Research and Industrial Applications*, pages 125–141, Arc et Senans, France, May 20–22, 1992. IOS Press, Amsterdam, Netherlands.

[80] David E. Culler, Seth C. Goldstein, Klaus E. Schauser, and Thorsten von Eicken. TMA – a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, July 1993.

[81] William J. Dally and Andrew A. Chien. Object-oriented concurrent programming in CST. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 28–31, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[82] Andrew Davison. A survey of logic programming-based object-oriented languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 42–106. MIT Press, 1993.

[83] Peter de Jong. Concurrent organizational objects. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 40–44, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[84] D. Decouchant. Design of a distributed object manager for the Smalltalk-80 system. In *Proc. of OOPSLA'86, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 444–452, Portland, Oregon, September 29 – October 2 1986. ACM SIGPLAN Notices 21(11).

[85] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill, and X. Rousset de Pina. A synchronization mechanism for typed objects in a distributed system. *ACM SIGPLAN Workshop on Concurrent Object-Based Language Design, in ACM SIGPLAN Notices*, 24(4):105–107, April 1989.

[86] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[87] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*. Academic Press, New York, 1968.

[88] E. W. Dijkstra. The structure of the 'THE' multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[89] E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[90] Takanobu Baba et al. A network-topology independent task allocation strategy for parallel computers. In *Proc. Supercomputing '90*, pages 878–887, 1990.

[91] Ian Foster. *Designing and Building Parallel Programs*, pages 167–205. Addison-Wesley, Reading, Mass., 1994.

[92] Arne Frick, W. Zimmer, and Wolf Zimmermann. On the design of reliable libraries. In *TOOLS 17 – Technology of Object-Oriented Programming*, pages 13–23, Santa Barbara, CA, August 2–4, 1995. Prentice Hall, Englewood Cliffs, New Jersey.

[93] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Implementation*. Addison-Wesley, Reading, Mass., 1983.

[94] Yvon Gourhant and Marc Shapiro. FOG/C++: a fragmented-object generator. In *C++ Conf.*, pages 63–74, San Francisco, CA, April 1990.

[95] J. E. Grass and R. H. Campbell. Mediators: a synchronization mechanism. In *Proc. of the 6th Int. Conf. on Distributed Computing Systems*, pages 468–477, Cambridge, MA, May 19–23, 1986. IEEE Comput. Soc. Press.

[96] Andrew S. Grimshaw. Easy to use object-oriented parallel programming. *IEEE Computer*, 26(5):39–51, May 1993. Also University of Virginia, Charlottesville, VA, Technical Report CS-92-32.

[97] Andrew S. Grimshaw. The Mentat computation model – data-driven support for object-oriented parallel processing. Technical Report CS-93-30, University of Virginia, Charlottesville, VA, May 1993.

[98] Andrew S. Grimshaw and V. E. Vivas. FALCON: A distributed scheduler for MIMD architectures. In *Proc. of the Symp. on Experiences with Distributed and Multiprocessor Systems*, pages 149–163, Atlanta, GA, March 1991.

[99] Andrew S. Grimshaw, Jon B. Weissan, and W. Timothy Strayer. Portable run-time support for dynamic object-oriented parallel processing. Technical Report CS-93-40, University of Virginia, Charlottesville, VA, July 1993.

[100] Andrew S. Grimshaw, Jon B. Weissman, Emily A. West, and Ed C. Loyot, Jr. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 21(3):257–270, June 1994. Also University of Virginia, Charlottesville, VA, Technical Report CS-92-43.

[101] Mentat Research Group. Mentat 2.5 programming language reference manual. Technical report, University of Virginia, Charlottesville, VA, 1995.

[102] R. Guerraoui. Dealing with atomicity in object-based distributed systems. *OOPS Messenger*, 3(3):10–13, July 1992.

[103] L. Gunaseelan and R. J. LeBland. Distributed Eiffel: A language for programming multigranular distributed objects. In *Proc. of the 4th Int. Conf. on Computer Languages (IEEE)*, San Francisco, CA, April 1992.

[104] B. K. Haddon. Nested monitor calls. *Operating Systems Review*, 11(4):18–23, October 1977.

[105] Daniel Hagimont, P.-Y. Chevalier, A. Freyssinet, S. Krakowiak, S. Lacourte, J. Mossière, and X. Rousset de Pina. Persistent shared object support in the Guide system: Evaluation & related work. In *Proc. of OOPSLA'94, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 129–144, Portland, OR, October 23–27, 1994.

[106] F. Hamelin, J.-M. Jézéquel, and T. Priol. A multi-paradigm object oriented parallel environment. In H. J. Siegel, editor, *Proc. of the 8th Int. Parallel Processing Symp. IPPS'94*, Cancún, Mexico, April 1994. IEEE Computer Society Press.

[107] P. Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.

[108] P. Brinch Hansen. Concurrent programming concepts. *ACM Computing Surveys*, 5(4):223–245, December 1973.

[109] P. Brinch Hansen. *Operating System Principles*. Prentice Hall, Englewood Cliffs, New Jersey, 1973.

[110] C. L. Hartley and V. S. Sunderam. Concurrent programming with shared objects in networked environments. In *Proc. of the 7th Int. Parallel Processing Symp.*, pages 471–478, Los Angeles, April 1993.

[111] Ernst A. Heinz. Modula-3*: An efficiently compilable extension of Modula-3 for explicitly parallel problem-oriented programming. In *Joint Symp. on Parallel Processing*, pages 269–276, Waseda University, Tokyo, May 17–19, 1993.

[112] Heinz-Peter Heinzle, Henri E. Bal, and Koen Langendoen. Implementing object-based distributed shared memory on Transputers. In A. De Gloria, M. R. Jand, and D. Marini, editors, *Transputer Applications and Systems '94*. IOS Press, 1994.

[113] Martin Henz. The Oz notation. Technical report, DFKI, German Research Center for Artificial Intelligence, Saarbrücken, Germany, 1994.

[114] B. Hindel. An object-oriented programming language for distributed systems: HERAKLIT. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 114–116, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[115] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*, pages 61–71. Academic Press, New York, 1972.

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

[117] W. Horwat, A. A. Chien, and W. J. Dally. Experience with CST: programming and implementation. In *Proc. of the ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation PLDI*, pages 101–109, Portland, OR, June 21–23, 1989. ACM SIGPLAN Notices 24(7).

[118] Kaoru Hosokawa and Hiroaki Nakamura. Concurrent programming in COB. In A. Yonezawa and T. Ito, editors, *Proc. of the Japan/UK Workshop on Concurrency: Theory, Language and Architecture*, pages 142–156, Oxford, UK, September 25–27. 1989. Springer-Verlag Berlin, Heidelberg, New York.

[119] Chris Houck and Gul Agha. HAL: A high-level Actor language and its distributed implementation. In *21st Int. Conf. on Parallel Processing, ICPP '92*, volume II, pages 158–165, St. Charles, IL, August 1992.

[120] Jin H. Hur and Kilnam Chon. Overview of a parallel object-oriented language CLIX. In *Proc. of ECOOP'87 – European Conf. on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, pages 265–273, Paris, France, June 15–17, 1987. Springer-Verlag Berlin, Heidelberg, New York.

[121] Norman C. Hutchinson, Rajeandra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald programing lanuage report. Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, October 1987.

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

[123] Yutaka Ishikawa, Atsushi Hori, Hiroki Konaka, Munenori Maeda, and Takashi Tomokiyo. MPC++: A parallel programming language and its parallel objects support. In *Proc. of the Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pages j1–j5, OOPSLA'93, Washington D.C., September 27, 1993.

[124] J.-M. Jézéquel. EPEE: an Eiffel environment to program distributed memory parallel computers. In *Proc. of ECOOP'92 – European Conf. on Object-Oriented Programming*, number 615 in Lecture Notes in Computer Science, pages 197–212, Utrecht, The Netherlands, June 29 – July 3, 1992. Springer-Verlag Berlin, Heidelberg, New York.

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

[126] J.-M. Jézéquel. Transparent parallelisation through reuse: between a compiler and a library

approach. In *Proc. of ECOOP'93 – 7th European Conf. on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 384–405, Kaiserslautern, Germany, July 26–30, 1993. Springer-Verlag Berlin, Heidelberg, New York.

[127] Eric Jul. Migration of light-weight processes in Emerald. *IEEE Operating Sys. Technical Committee Newsletter, Special Issue on Process Migration*, 3(1):25–30, 1989.

[128] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[129] Dennis Kafura. Concurrent object-oriented real-time systems research. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 203–205, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[130] Dennis Kafura and Greg Lavender. Recent progress in combining Actor based concurrency with object-oriented programming. In *Proc. of ECOOP OOPSLA'90, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 55–58, Ottawa, Canada, October 21–25, 1990.

[131] Dennis Kafura and K. H. Lee. ACT++: Building a concurrent C++ with Actors. *Journal of Object Oriented Programming*, 3(1):25–37, May 1990.

[132] Dennis Kafura, Manibrata Mukherji, and Greg Lavender. ACT++ 2.0: A class library for concurrent programming in C++ using Actors. *Journal of Object Oriented Programming*, 6(6):47–55, October 1993.

[133] Dennis G. Kafura and Keung Hae Lee. Inheritance in Actor based concurrent object-oriented languages. In *ECOOP'89 – European Conf. on Object-Oriented Programming*, pages 131–145. Cambridge University Press, 1989.

[134] Gail E. Kaiser, Wenwey Hseush, James C. Lee, Shyhtsun F. Wu, Esther Woo, Eric Hilsdale, and Scott Meyer. MeldC: A reflective object-oriented coordination language. Technical Report CUCS-001-93, Dept. of Computer Science, Columbia University, New York, January 1993.

[135] Gail E. Kaiser, Wenwey Hseush, Steven S. Popovich, and Shyhtsun F. Wu. Multiple concurrency control policies in an object-oriented programming system. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 195–210. MIT Press Cambridge, Massachusetts, London, England, 1993.

[136] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proc. of OOPSLA'93, 8th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–109, Washington D.C., 26 September – 1 October, 1993. ACM SIGPLAN Notices 28(10).

[137] Vijay Karamcheti and Andrew Chien. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Proc. of ACM Supercomputing'93*, pages 598–607, Portland, Oregon, November 15–19, 1993.

[138] Murat Karaorman and John Bruno. Design and implementation issues for object-oriented concurrency. In *Proc. of the Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pages m1–m9, OOPSLA'93, Washington D.C., September 27, 1993.

[139] Murat Karaorman and John Bruno. Introduction of concurrency to a sequential language. *Communications of the ACM*, 37(9):103–116, September 1993.

[140] J. L. W. Kessels. An alternative to event queues for synchronization in monitors. *Communications of the ACM*, 20(7):500–503, July 1977.

[141] WooYoung Kim and Gul Agha. Compilation of a highly parallel Actor-based language. In *Proc. of the 5th Int. Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 1–12, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag Berlin, Heidelberg, New York.

[142] K. D. Kooper and K. Kennedy. Fast interprocedural alias analysis. In *Proc. of the 16th ACM Symp. on Principles of Programming Languages*, pages 49–59, Austin, TX, January 1989.

[143] Tim Korson and John D. McGregor. Understanding object-oriented: A unifying paradigm.

Communications of the ACM, 33(9):40–60, September 1990.

[144] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, and X. Rousset de Pina. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object Oriented Programming*, 3(3):11–22, September/October 1990.

[145] J. Kramer, J. Magee, M. Sloman, N. Dulay, S. Cheung, S. Crane, and K. Twidle. An introduction to distributed programming in Rex. In *Procedings of the 1991 Esprit Conf.*, pages 207–222, Brussels, Belgium, November 1991.

[146] Serge Lacourte. Exceptions in Guide, an object-oriented language for distributed applications. In *Proc. of ECOOP'91 – European Conf. on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, pages 268–287, Geneva, Switzerland, July 15–19, 1991. Springer-Verlag Berlin, Heidelberg, New York.

[147] Chris Laffra and Jan van den Bos. Constraints in concurrent object-oriented environments. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 64–67, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[148] Chris Laffra and Jan van den Bos. Propagators and concurrent constraints. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 68–72, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[149] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proc. of the SIGPLAN Conf. on Programming Language Design and Implementation*, pages 56–67, Albuquerque, New Mexico, June 1993. ACM SIGPLAN Notices 28(6).

[150] J. Larus. C**: A large-grain object-oriented, data-parallel programming language. In *Proc. of the 5th Int. Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 326–341, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag Berlin, Heidelberg, New York.

[151] James R. Larus, Brad Richards, and Guhan Viswanathan. C**: A large-grain object-oriented, data-parallel programming language. Technical Report UWTR-1126, Computer Science Department, University of Wisconsin, Madison, November 1992.

[152] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory system support for parallel language implementation. In *Proc. of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS'94*, pages 208–218, October 4–7, 1994. Also available as Computer Science Department, University of Wisconsin, Madison, Technical Report TR1237.

[153] Rodger Lea, Christian Jacquemot, and Eric Pillevesse. COOL: System support for distributed programming. *Communications of the ACM*, 36(9):37–46, September 1993.

[154] Rodger Lea and James Weightman. Supporting object oriented languages in an distributed environment: The COOL approach. In *Proc. of Conf. on Technology of Object-Oriented Languages and Systems, TOOLS USA'91*, Santa Barbara, August 3–6, 1991. Prentice Hall, Englewood Cliffs, New Jersey.

[155] Jeng Kuen Lee and Yunn-Yen Chen. Compiler and library support for aggregate object communications on distributed memory machines. In *Proc. of the Workshop on Efficient Implementation of Concurrent Object-Oriented Languages*, pages d1–d10, OOPSLA'93, Washington D.C., September 27, 1993.

[156] Jenq Kuen Lee and Dennis Gannon. Object oriented parallel programming – experiments and results. In *Proc. of Supercomputing'91*, pages 273–282, Albuquerque, NM, November 18–22, 1991.

[157] Y. S. Lee, J. H. Huang, and F. J. Wang. A distributed Smalltalk based on process-object model. In G. J. Knafl, editor, *Proc. of the 15th Annual Int. Computer Software and Applications Conf.*, pages 465–471, Tokyo, Japan, September 11–13, 1991. IEEE Comput. Soc. Press.

[158] A. Lister. The problem of nested monitor calls. *Operating Systems Review*, 11(3):5–7, July 1977.

[159] Klaus-Peter Löhr. Concurrency annotations. *ACM SIGPLAN Notices*, 27(10):327–340, October 1992.

[160] Klaus-Peter Löhr. Concurrency annotations for reusable software. *Communications of the ACM*, 36(9):81–89, September 1993.

[161] Cristina Videira Lopes and Karl J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In Mario Tokoro and Remo Pareschi, editors, *Proc. of the 8th European Conf. on Object-Oriented Programming, ECOOP'94*, number 821 in Lecture Notes in Computer Science, pages 81–99, Bologne, Italy, July 4–8, 1994. Springer-Verlag Berlin, Heidelberg, New York.

[162] S.A. MacKay, W.M. Gentleman, D.A. Stewart, and M. Wein. Harmony as an object-oriented operating system. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 209–211, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[163] Ole Lehrmann Madsen. Building abstractions for concurrent object-oriented programming. Technical report, Computer Science Department, Aarhus University, Denmark, February 1993.

[164] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Mygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, Mass., 1993.

[165] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*. IEEE Computer Society Press, July 1993.

[166] Carl Manning. A peek at Acore, an Actor core language. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 84–86, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[167] Katsumi Maruyama and Nicolas Raguideau. Concurrent object-oriented language COOL. *ACM SIGPLAN Notices*, 29(9):105–114, September 1994.

[168] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proc. of OOPSLA'92, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Vacnouver, Canada, October 18–22, 1992. ACM SIGPLAN Notices 27(10).

[169] Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa. An object-oriented concurrent reflective language for dynamic resource management in highly parallel computing. In *IPSJ SIG Notes*, volume 94-PRG-18, pages 57–64, 1994.

[170] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press Cambridge, Massachusetts, London, England, 1993.

[171] Jeff McAffer and John Duimovich. Actra – an industrial strength concurrent object oriented programming system. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 82–84, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[172] Paul L. McCullough. Transparent forwarding: First steps. In *Proc. of OOPSLA'87, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 331–341, Orlando, Florida, October 4–8, 1987. ACM SIGPLAN Notices 22(12).

[173] Ciaran McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, Department of Computer Science, Trinity College, Dublin 2, Ireland, October 1994.

[174] Ciaran McHale, Bridget Walsh, Seán Baker, and Alexis Donnelly. Scheduling predicates. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proc of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, pages 177–193, Geneva, Switzerland, July 15–16, 1991. Springer-Verlag Berlin, Heidelberg, New York.

[175] Piyush Mehrotra and John Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, November 1987.

[176] Piyush Mehrotra and John Van Rosendale. Concurrent object access in BLAZE 2. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 40–42, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[177] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[178] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

[179] Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, September 1993.

[180] J. Elliot B. Moss and Walter H. Kohler. Concurrency features for the Trellis/Owl language. In *Proc. of ECOOP'87 – European Conf. on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, pages 171–180, Paris, France, June 15–17, 1987. Springer-Verlag Berlin, Heidelberg, New York.

[181] Claudio Nascimento and Jean Dollimore. Behavior maintenance of migrating objects in a distributed object-oriented environment. *IEEE Computer*, 25(9), September 1992.

[182] Oscar Nierstrasz. Active objects in Hybrid. In *Proc. of OOPSLA'87, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 243–253, Orlando, Florida, October 4–8, 1987. ACM SIGPLAN Notices 22(12).

[183] Oscar Nierstrasz. A tour of Hybrid: A language for programming with active objects. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 167–182. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[184] Mark Nuttal. A brief survey of systems providing process or object migration facilities. *Operating Systems Review*, 28(4):64–80, October 1994.

[185] Kazuhiro Ogata, Satoshi Kurihara, Mikio Inari, and Norihisa Doi. The design and implementation of HoME. In *Proc. of the ACM SIGPLAN Conf. on Programming Languages, Design and Implementation, PLDI'92*, pages 44–54, San Francisco, CA, June 17–19 1992.

[186] Ronald A. Olsson, Gregory R. Andrews, Michael H. Coffin, and Gregg M. Townsend. SR – a language for parallel and distributed programming. Technical Report TR 92-09, Dept. of Computer Science, University of Arizona, Tucson, March 1992.

[187] Joseph Pallas and David Ungar. Multiprocessor Smalltalk a case study of a multiprocessor-based programming environment. In *Proc. of SIGPLAN Conf.*, pages 268–277, 1988.

[188] M. Papathomas. Concurrency issues in object-oriented programming languages. In D. Tsichritzis, editor, *Object Oriented Development*, pages 207–245. University of Geneva, Switzerland, 1989.

[189] Michael Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming*. PhD thesis, Université de Genève, Départment d'Informatique, January 1992.

[190] *Proc. of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*. ACM SIGPLAN Notices 28(12), San Diego, CA, May 17–18, 1993.

[191] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[192] D. L. Parnas. The non-problem of nested monitor calls. *Operating Systems Review*, 12(1):12–14, January 1978.

[193] Michael Philippsen. Imperative concurrent object-oriented languages: An annotated bibliography. Technical Report TR-95-049, International Computer Science Institute, Berkeley, August 1995.

[194] Michael Philippsen and Ernst A. Heinz. Automatic synchronization elimination in synchronous foralls. In *Frontiers '95:The 5th Symp. on the Frontiers of Massively Parallel Computation*, pages 350–357, Mc Lean, VA, February 6–9, 1995.

[195] Michael Philippsen and Walter F. Tichy. Modula-2* and its compilation. In *1st Int. Conf. of the Austrian Center for Parallel Computation, Salzburg, Austria, 1991*, pages 169–183. Springer Verlag, Lecture Notes in Computer Science 591, 1992.

[196] John Plevyak, Xingbin Zhang, and Andrew A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proc. of the 22nd Annual ACM SIGACT–SIGPLAN Symp. on Principles of Programming Languages POPL'95*, pages 311–321, San Francisco, CA, January 22–25, 1995.

[197] Steven S. Popovic, Gail E. Kaiser, and Shyhtsum F. Wu. MELDing transactions and objects. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 94–98, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[198] Roland Pozo. A stream-based interface in C++ for programming heterogeneous systems. In *Proc. of the CRNS-NSF Workshop on Environment and Tools for Parallel Scientific Computing*, pages 162–177, Saint Hilaire du Touvet, France, September 7–8, 1992. Elsevier, Advances in Parallel Computing, Vol. 6, 1993.

[199] Myra Jean Prelle, Ann M. Wollrath, Thomas J. Brando, and Edward H. Bensley. The impact of selected concurrent language constructs on the SAM run-time system. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 99–103, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[200] R. S. Pressmann. *Software Engineering*. McGraw-Hill Book Company, New York, 1987.

[201] Donna S. Reese and Ed Luke. Object oriented Fortran for development of portable parallel programs. In *Proc. of the 3rd IEEE Symp. on Parallel and Distributed Processing*, pages 608–615, Dallas, Texas, December 2–5, 1991.

[202] M. Riveill. An overview of the Guide language. In *2nd Workshop on Objects in Large Distributed Applications*, Vancouver (Canada), 18 October 1992.

[203] Hayssam Saleh and Philippe Gautron. A concurrency control mechanism for C++ objects. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proc. of the ECOOP'91 Workshop on object-based concurrent computing*, pages 195–210, Geneva, Switzerland, July 15–16, 1991. Springer-Verlag Berlin, Heidelberg, New York.

[204] Hayssam Saleh and Philippe Gautron. A system library for C++ distributed applications on Transputer. In *Proc. of the 3rd Int. Conf. on Applications of Transputers*, pages 638–643. IOS Press, Amsterdam, Netherlands, August 28–30, 1991.

[205] Michele Di Santo and Giulio Iannello. Implementing actor-based primitives on distributed-memory architectures. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 45–49, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[206] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *Proc. of OOPSLA'86, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 9–16, Portland, Oregon, September 29 – October 2, 1986. ACM SIGPLAN Notices 21(11).

[207] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis – object-based environment: Language reference manual. Technical Report DEC-TR-372, Eastern Research Lab, DEC, Hudson, Massachusetts, November 1985.

[208] Marcel Schelvis and Eddy Bledoeg. The implementation of a Distributed Smalltalk. In *Proc. of the European Conf. on Object-Oriented Programming, ECOOP'88*, number 322 in Lecture Notes in Computer Science, pages 212–232, Oslo, Norway, August 15–17, 1988. Springer-Verlag Berlin, Heidelberg, New York.

[209] Heinz W. Schmidt. Data parallel object-oriented programming. In *Proc. of the 5th Australian Supercomputer Conf.*, pages 263–272, Melbourne, December 1992.

[210] Robert J. Smith. Experimental systems kit – final project report. Technical report, Microelectronics and Computer Technology Corporation, MCC, Austin, Texas, March 1991.

[211] Gert Smolka. The definition of kernal Oz. Technical report, DFKI, German Research Center for

Artificial Intelligence, Saarbrücken, Germany, 1994.

[212] Gert Smolka. An Oz primer. Technical report, DFKI, German Research Center for Artificial Intelligence, Saarbrücken, Germany, April 1995.

[213] Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*, pages 27–48. The MIT Press, 1995.

[214] A. Snyder. Encapsulation and inheritance. In *Proc. of OOPSLA'86, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–45, Portland, Oregon, September 29 – October 2 1986. ACM SIGPLAN Notices 21(11).

[215] Jan van der Spek. POOL-X and its implementation. In Pierre America, editor, *Parallel Database Systems. PRISMA Workshop*, pages 309–344, Noordwijk, The Netherlands, September 24–26, 1990. Springer-Verlag Berlin, Heidelberg, New York.

[216] Kazunori Takashio and Mario Tokoro. DROL: An object-oriented programming language for distributed real-time systems. In *Proc. of OOPSLA'92, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 276–294, Vacnouver, Canada, October 18–22, 1992. ACM SIGPLAN Notices 27(10).

[217] Hidehiko Tanaka. A parallel object oriented language FLENG++ and its control system on the parallel machine PIE64. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language and Architecture. Japan/UK Workshop Proc.*, pages 157–172. Springer-Verlag Berlin, Heidelberg, New York, 1991.

[218] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henry E. Bal. Parallel programming using shared objects and braodcasting. *IEEE Computer*, 25(18):10–19, August 1992.

[219] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language – its design and implementation. In G. Blelloch, M. Chandy, and S. Jagannathan, editors, *Proc. of the DIMACS workshop on Specification of Parallel Algorithms*, 1994.

[220] Thinking Machines Corporation, Cambridge, Massachusetts. **Lisp Reference Manual, Version 5.0*, 1988.

[221] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Language Reference Manual*, April 1991.

[222] David A. Thomas, Wilf R. LaLonde, John Duimovich, Michael Wilson, Jeff McAffer, and Brian Barry. Actra - a multitasking/multiprocessing Smalltalk. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 87–89, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[223] Michael D. Tiemann. Solving the RPC problem in GNU C++. Technical Report ESKIT-285-88, Microelectronics and Computer Technology Corporation, MCC, Austin, Texas, 1988.

[224] Chris Tomlinson, Won Kim, Marek Scheevel, Vineet Singh, Becky Will, and Gul Agha. Rosette: an object-oriented concurrent system architecture. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 91–93, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[225] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with Enabled-sets. In *Proc. of OOPSLA'89, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 103–112, New Orleans, Louisiana, October 1–6, 1989. ACM SIGPLAN Notices (24)10.

[226] Christine Tomlinson, Mark Scheevel, and Vineet Singh. *Report on Rosette 1.1*, August 1991. Object-Oriented and Distributed Systems Laboratory, Microelectronics and Computer Technology Corp., MCC.

[227] Rajiv Trehan, Nobuyuki Sawashima, Akira Morishita, Ichiro Tomoda, Toru Imai, and Ken ichi Maeda. Concurrent object oriented 'C' (cooC). *ACM SIGPLAN Notices*, 28(2):45–52, February 1993.

[228] Chau-Wen Tseng. Compiler optimizations for eliminating barrier synchronization. In *5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP*, pages 144–155, Santa Barbara, CA, July 19–21 1995.

[229] Louis H. Turcotte. A survey of software environments for exploiting network computing resources. Technical report, Mississippi State University, June 11, 1993.

[230] Minoru Uehara and Mario Tokoro. An adaptive load balancing method in the computational field model. In *Proc. of ECOOP OOPSLA'90 Workshop on object-based concurrent programming*, pages 109–113, Ottawa, Canada, October 21–22, 1990. OOPS Messenger 2(2) April 1991.

[231] Jean Vaucher, Guy Lapalme, and Jacques Malenfant. SCOOP – structured concurrent object-oriented prolog. In *ECOOP'88 – European Conf. on Object-Oriented Programming*, pages 191–210, Oslo, Norway, August 15–17, 1988. Springer-Verlag Berlin, Heidelberg, New York.

[232] Peter Wegner. Dimensions of object.based language design. In *Proc. of OOPSLA'87, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 168–182, Orlando, Florida, October 4–8, 1987. ACM SIGPLAN Notices 22(12).

[233] Peter Wegner. Tradeoffs between reasoning and modeling. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 23–40. MIT Press, 1993.

[234] Emily A. West. *Combining Control and Data Parallelism: Data Parallel Extensions to the Mentat Programming Language.* PhD thesis, University of Virginia, Department of Computer Science, May 1994. Available as technical report CS-94-16.

[235] Emily A. West and Andrew S. Grimshaw. Braid: Integrating task and data parallelism. In *Frontiers '95: The 5th Symp. on the Frontiers of Massively Parallel Computation*, pages 211–219, McLean, VA, February 6–9, 1995.

[236] R. H. H. Wester and B. J. A. Hulshof. The POOMA operating system. In Pierre America, editor, *Parallel Database Systems. PRISMA Workshop*, pages 396–323, Noordwijk, The Netherlands, September 24–26, 1990. Springer-Verlag Berlin, Heidelberg, New York.

[237] H. Wettstein. The problem of nested monitor calls revisited. *Operating Systems Review*, 12(1):19–23, January 1978.

[238] R. Winder, G. Roberts, and M. Wei. Co-SIDE and parallel object-oriented languages. In *Addendum to the Proc. of OOPSLA'92, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–213, Vacnouver, Canada, October 5–10, 1992.

[239] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.

[240] Barbara Wyatt, Krishna Kavi, and Steve Hufnagel. Parallelism in object-oriented languages: a survey. *IEEE Computer*, 11(6):56–66, November 1992.

[241] Gao Yaoqing and Yuen Chung Kwong. A survey of implementations of concurrent, parallel and distributed Smalltalk. *ACM SIGPLAN Notices*, 28(9):29–35, September 1993.

[242] Stephen S. Yau, Xiaoping Jia, Doo-Hwan Bae, Madhan Chidambaram, and Gilho Oh. An object-oriented approach to software development for parallel processing systems. In G. J. Knafl, editor, *Proc. of the 15th Annual Int. Computer Software and Applications Conf.*, pages 453–5–8, Tokyo, Japan, September 11–13, 1991. IEEE Comput. Soc. Press.

[243] Yasuhiko Yokote and Mario Tokoro. The design and implementation of ConcurrentSmalltalk. In *Proc. of OOPSLA'86, Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 331–340, Portland, Oregon, September 29 – October 2 1986. ACM SIGPLAN Notices 21(11).

[244] Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System – theory, language, programming, implementation, and application.* Computer System Series. MIT Press Cambridge, Massachusetts, London, England, 1990.

[245] Kaoru Yoshida and Takashi Chikayama. A'UM = stream+object+relation. In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 55–58, San Diego, CA, September 26–27, 1988. ACM SIGPLAN Notices 24(4).

[246] Tsutomu Yoshinaga and Takanobu Baba. A parallel object-oriented language A-NETL and its programming environment. In G. J. Knafl, editor, *Proc. of the 15th Annual Int. Computer*

Software and Applications Conf., pages 459–464, Tokyo, Japan, September 11-13, 1991. IEEE Comput. Soc. Press.