VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Abstract Concurrency

Ewan Tempero

Technical Report CS-TR-91/5
October 1991

Abstract

Two favourite buzz-words in current programming language research are *object-oriented* and *parallelism* and so naturally there has been a lot of interest in merging the two concepts. In this paper I present my attempt. I examine several typical problems and make some observations about the natural concurrency that occurs in them. From these observations, I develop, from first principles, an appropriate model of concurrent programming that is best suited to object-oriented languages. I compare my model with that found in existing languages.

Publishing Information

This report is a revised version of that which appeared in *Proceedings of the 12th New Zealand Computer Conference*, Dunedin. 14–16 August 1991.

Author Information

Ewan Tempero graduated from the University of Otago with a BSc (Honours) in mathematics in 1983 and gained a PhD in computer science from the University of Washington, Seattle in 1990. He is currently a lecturer in computer science at Victoria University of Wellington.

1 Introduction

Object-oriented programming languages have become very popular due, in part, to their ability to provide code reuse and the improvement in code management that results but also because they provide very good support for application design techniques based on modularization and abstraction. An active area of research is investigating ways to provide for concurrent programming for object-oriented programming languages and it is this area that this paper addresses.

While many models for concurrency have been introduced for object-oriented languages, for the most part they have merely been extensions to existing languages or continued the debates on appropriate concurrency mechanisms begun in the non-object-oriented world [Lim and Johnson, 1988]. My approach is to examine the programming model that object-oriented languages enforces and determine the best way to modify that model to allow concurrency of an application to be expressed.

The goal of this approach is to avoid the problems that arise when the programming model enforced by the implementation language is different from the design technique. If such a mismatch exists, then some programming effort must be expended to correct the problem. This complicates the implementation and so increases the cost. An extreme example of a mismatch is when the design assumes the existence of a record-like structure, but the implementation language only provides arrays. More subtle mismatches can arise when dealing with concurrency.

The paper is organized as follows. I will begin by describing what building an application with objects means and how this relates to the design of an application. These first principles lead to a question which I consider fundamental for developing a concurrency model for object-oriented languages. In section 3, I present some example applications from which I try to find an answer to this fundamental question. From this discussion, I derive my concurrency model in section 4 and section 5 discuss some of its consequences. Section 6 compares this model to models in other languages and in section 7, I mention some issues and non-issues relating to this work. Finally in section 8 I present my conclusions.

2 First Principles

There are several reasons why a programming language might be developed, and different reasons often result in different emphasis. Let me say now what my reasons are so as to provide the proper context for this discussion and prevent misunderstandings due to differing goals. My main goal is that of correct implementation of an application design. Since the programming language, or rather, the programming model it supports, tends to influence the final stages of a design [Levy and Tempero, 1991], to achieve this goal, it is necessary to provide a programming model that least obstructs the design process. A secondary goal is to provide a programming model that clearly reflects an application's design. This not only makes it easy to confirm that the implementation meets the design, but also improves software maintainability.

I will often use the terms *process*, *thread*, and concurrency. I will use process when I am describing an entity that can perform actions independently from other entities and thread when I am concerned with a single thread of control. Thus it is possible that a process may contain several threads but not vice versa. Concurrency refers to the potential for more than one threads of control to execute simultaneously.

My basic premis is that an application is modeled as a dynamic set of interacting objects. An object consists at least of an ID (or reference) by which it may be identified to other entities in the system, some local state (also modeled as a set of objects), and a public interface which gives defines the external behaviour of the object. Specifically, the interface describes how the state of the object can be reported and manipulated. This is the only way the state of the object can be accessed, that is, an object *encapsulate* its state. Access through this interface is called *communication*. An object can only communicate with another object if it knows the ID of (has a reference to) that other object. An object records the IDs of objects it knows in its local state. Many object-oriented languages model the object interface as a set of *messages* that the object understands. However "message" has many interpretations, often including some notion of asynchrony. To avoid confusion, I will consider the interface of an object to be a set of *operations* and talk about *invoking* an operation.

Each object represents some logical entity in the application being modeled. If an entity is composed of several other enties (for example, a table consists of a surface and four legs), the object's local state will represent those components. Thus an application consists of a set of objects that are related by the "know ID" property.

The design of any application typically involves identifying the *abstractions* that make up the application. An abstraction consists of some behaviour which often depends on some state. This is exactly what an object is and so an object can be considered to be a concrete representation of an abstraction.

Abstractions can occur at different levels. For example, a particular version of the abstraction *waiting-list* may include the abstraction of a *FIFO queue*; a FIFO queue may be realized as a *linked-list* or *array*, which are abstractions in their own right. The translation of this hierarchy of abstractions into objects could be as follows: First there is an object representing the top level abstraction, waiting-list. The local state of this object includes a reference to an object that represents a FIFO queue and the local state of that object includes references that are used to implemented a linked list. Thus the levels of abstraction translate into levels of objects.

Given that objects are just concrete representations of abstractions, to develop the most appropriate model for concurrent programming in object-oriented languages, the fundamental question is then:

> *What natural forms of concurrency exist in any abstraction?*

To answer this question, I will now examine some, hopefully representative, problems to determine what *abstract concurrency* must be expressed by a concurrent programming model.

3 Examples with Concurrency

In this section I give three examples of problems a programmer may want to solve. The three problems are: shared buffers, the classical concurrency example, a name server, an example of the client/server programming paradigm, and a computerized whiteboard, an example of a non-trivial application. I will describe and discuss what natural parallelism each problem exhibits.

3.1 Shared Buffers

A standard concurrency example from operating systems is the Producer/Consumer problem. The simplest form of this problem consists of two processes, the Producer and the Consumer. The Producer generates items of interest to the Consumer and places them in a finite-sized buffer. The Consumer removes these items when it needs them. The problem is to synchronize the processes' access to the buffer so that the Producer will wait while the buffer is full and the Consumer will wait while the buffer is empty.

The natural concurrency of this example is the two processes; each process has one thread of control. Although there are three entities in this system, the two processes and the buffer, only the processes act as independent agents. While the buffer's state must be modified when the processes act, performing the modification is not an independent action so there is no need for the buffer to be "multi-threaded." In fact, it is useful to think of the processes "lending their threads" to the buffer to make the modification.

3.2 Name Server

A Name Server maintains a set of names and attributes of those names. Users mainly query it to find attributes of a name, or names having certain attributes, however some modifications may be done occasionally. If all accesses are queries (that is, read-only) then they can all execute simultaneously. Thus the number of simultaneously running threads can be as many as there are users. One way to represent the relationships between names and attributes is as a tree. The names can be partitioned according to some attributes (for example, `machine name`) which is represented by branching at the higher levels and then other attributes can be attached to the name (for example, `internet address`). This representation allows simultaneous modifications at points in the tree that are not related (ancestors or descendents).

In fact, this example is really just a generalized form of the shared buffers example. Here, the "buffer" is more complex, allowing multiple simultaneous queries and updates. The natural concurrency is exactly the number of entities that wish to access the Name Server.

3.3 Computerized Whiteboard

The idea behind a computerized whiteboard is that each user who wants to use the whiteboard can open a view onto the whiteboard. Use of the whiteboard consists of freehand drawing and erasing. A whiteboard is a fairly informal method of communication so there is very little synchronization that has to be provided by the system. Any useful sharing of the whiteboard will require some protocol followed by the users, as is the case with real whiteboards. Each user should see a consistent view of the whiteboard. This implies that each time a user changes the current state of the whiteboard, all other users must be notified of the change.

The difference between this example and the Name Server is that the interaction is more complex. The Name Server just had to service requests, whereas the whiteboards must also notify users of changes.

The natural concurrency in this application is the users. Each user acts independently and possibly simultaneously. There is also some concurrency in the whiteboard itself; when a change is made by a user, the other users must be notified of the change and it seems natural to do this to all simultaneously.

The user, or rather, the application's representation of the user, must process input commands from the human, and display the current state of the whiteboard. The state of the whiteboard changes when the user it is associated with makes a change, or there is notification of a change by another user. In neither case is another thread needed (or appropriate); in the former case the thread that updates the display is the one that processes inputs and in the latter case, the thread comes from the whiteboard.

4 Concurrency Model

The examples suggest that there are two forms of abstract concurrency. The first is a single thread of control, such as that for the Producer or Consumer. The second form is a set of homogeneous threads, that is threads all doing the same thing, such as notifying a set of users of changes in the computerized whiteboard.

Any model of concurrent programming for object-oriented languages must provide a mechanism for representing both forms of abstract concurrency. The question now is how this should be done. It is instructive to consider the Producer process in the shared buffer example. In its simplest form, it is just a thread that alternates between creating items and putting them into the buffer. This implies that it is sufficient for the thread to be "freestanding," that is, not associated with any objects. Even if the Producer has some state, such as a record of the number of items produced, it is sufficient to simply allow threads to have local state modeled as a set of objects.

This argues that threads of control should be a completely separate concept from objects. This kind of talk upsets the object purists who insist that everything in a object-oriented system is an object. They argue that a separate concept of thread implies a different mechanism for creating and managing threads than there is for objects and it is preferable to keep a programming model as simple as possible.

However my complaint is that separating the thread from the object violates the idea that objects are implementations of abstractions. For example, consider the situation in which the Producer/Consumer system is a part of some larger system, one in which an Auditor must keep track of what items have been produced or consumed. In this case, the Producer is an abstraction that is just part of the abstraction of the whole system. If part of the Producer abstraction is a record of the number of items produced, then there must be some way for the Auditor to access this record. The only way such information can be accessed is if there is some way to query this part of the state; objects provide such a mechanism but threads do not. Thus the Producer cannot be just a thread, as described above. One could argue that a reference to the "items produced" object that is part of the thread's state could also be maintained outside of the thread, however this is does not conform to the notion of objects as implementations of abstractions. Thus threads should be associated with objects rather than exist as separate entities.

This raises the question of how many threads should be associated with an object. I claim that not more than one is needed despite my statement that there are two forms of abstract concurrency. The reason for this is there is always some abstraction which contains the homogeneous set of threads. For example, for the distributed whiteboard, the users that need to be notified of changes of the state of the whiteboard are just that, a set of users. This latter example appears to be something of a cop-out, however I think it illustrates my point; there is always some "top level" abstraction.

Finally, there is the problem of providing a homogeneous set of threads. I have argued that there are two forms of abstract concurrency but now only require that one thread be associated with each object so how is this problem to be resolved? The answer lies in the fact that what is really going on is that the same operation must be applied to a set of homogeneous objects. Thus, what is needed is a mechanism for allowing the simultaneous invocation of an operation to a set of objects.

5 Consequences

An application usually begins as an abstraction in the client's mind. The discussion above implies that, in the model I am proposing, there is only ever one thread of control in an application! However, as I have already explained, there are many levels of abstraction; the *state* of an abstraction can consist of many other abstractions, each with their associated thread of control. To see this, consider once again the Producer/Consumer example as a subsystem of a large application. For example, this behaviour is required for many kinds of I/O subsystems to smooth out bursty traffic. "I/O subsystem" is an abstraction, which, according to my model, should only have one thread of control. However, we already know that an implementation of the Producer/Consumer subsystem can have two threads in my model. Thus, the actual concurrency of an application depends on its actual implementation, as one would expect.

Another concern is whether this model will allow sufficient flexibility to describe different implementations with different kinds of concurrency. For example, the Name Server, instead of being implemented as a passive structure with the concurrency being supplied by the clients' threads, could instead be implemented using the "work pool" approach. In this approach, clients queue up to obtain service that is supplied by a fixed number of workers. The design of the Name Server now consists of the data structure containing the information and the set of workers. The amount of concurrency is now determined by the number of workers. This form of concurrency is easily represented in my model. There is a top level abstraction which is the Name Server. This abstraction's state consists of the information and the set of workers. Each worker is itself an abstraction—and so, in my model, may have a thread associated with it.

This discussion suggests that my model can provide both sufficient concurrency for any application without sacrificing the flexibility of begin able to provide this concurrency in different ways to take advantage of different situations.

6 Other Models

In this section, I look at several concurrent object-oriented languages. As I implied in the introduction, different goals can lead to different language design decisions and so arguing that the following languages do not satisfy my requirements may be unfair since these languages may have been designed with other goals in mind. However, I believe this list is representative of the current state-of-the-art as far as concurrent object-oriented programming goes, and thus argue that, if my goals are valid, then existing concurrent programming models are inadequate.

I will give a short description of the concurrency model that each language provides and then discuss what I perceive as being their shortcomings.

6.1 Smalltalk-80

Smalltalk-80[1] represents concurrency by the class *Process*. A instance of this class can be created by sending the message **fork** or **newProcess** to a block. Processes can have different priorities and are scheduled by an instance of the class *ProcessScheduler*. Synchronization is only done through semaphores, that is, there is no *join* operation. As a result, communication of results must be done through shared variables.

The main problem with Smalltalk-80 concurrency is that there is no real relationship between the threads created and any objects except block objects, which only represent the block abstraction and not any abstraction that is part of an application. Creating the Producer/Consumer system described in section 3 would require something like sending **new-Process** messages to blocks that describe the operation of the Producer and Consumer in the context of a variable that maintains the state of the shared buffer. If the Producer keeps a record of the number of items produced, there is no way to interrogate the Producer block for this information.

[1]Smalltalk-80 is a trademark of Xerox Corporation.

In general, this style of concurrent program is no better than using a **go to** construct. It does not impose any discipline on the use of threads so it is very easy to create many threads whose interaction is very complex and difficult to understand [Hoare, 1985].

6.2 ConcurrentSmalltalk

ConcurrentSmalltalk [Yokote and Tokoro, 1986, Yokote and Tokoro, 1987] is another attempt to add concurrency to Smalltalk. The main constructs provided are asynchronous message sending, atomic objects, and special objects, called *CBoxes*, for synchronization. The programmer can control the amount of concurrency in an application by choosing between synchronous and asynchronous message sending. An asynchronous message send will return a CBox and the result of the message send can be retrieved by sending a *receive* message to the CBox. Atomic objects have the property that messages sent to them are processed in FIFO order.

Yokote and Tokoro justify their concurrency model with the claim that the Smalltalk-80 model creates problems for the programmer because there are two mechanisms that must be dealt with, the object and the process, and that their model combines the notions of object and process. However I do not understand how allowing asynchronous message sends achieve this goal. It is not even clear what unifying objects and processes mean. Most definitions of "process" describe a sequence of instructions that are implemented serially whereas an object is some state and operations that can be performed on that state. It could be that Yokote and Tokoro want the only entities in the system to be objects however all they have really done, by introducing asynchronous message sends, is to make the independent threads of control implicit, instead of explicit as in the case with Smalltalk-80. I would argue that, rather than make the programming model easier, making the concurrency implicit makes it harder to reason about the behaviour of the software module, which conflicts with my goals. This argument is supported by the experience of members of the Eden project [Black, 1985].

Despite the arguments above, ConcurrentSmalltalk does allow easier implementations of some problems. The Producer/Consumer system can be described as a instance of a Producer/Consumer class that has two methods: Producer and Consumer. The methods are just loops that describe the behaviour of the Producer and Consumer. They can be started by way of the asynchronous message send mechanism. However, once again, there is no object corresponding to the Producer or Consumer abstractions.

6.3 Actors

The Actor model [Agha, 1986a, Agha, 1986b, Agha, 1988, Agha and Hewitt, 1987] consists of a set of actors communicating with each other. An actor consists of a *mail address* and a *behaviour*. An actor may send communications to any actor it has a mail address for. Communications sent to an actor are buffered in its mailbox, thus providing asynchronous communication. Whenever an actor receives a communication, it processes that message according to its current behaviour, and then creates a new behaviour to accept the next communication. An actor may create other actors and mail addresses can be freely communicated.

Agha claims that the concurrency is spawned in an actor system by sending more than one message in response to receiving a message. However, since actors processes the messages in their mail box sequentially, the amount of concurrency is actually related to the number of actors in the system.

The Actor programming model does appear, at first glance, to satisfy some of the requirements of my model. Each actor is very similar to an object; the communications the actor will accept correspond to the operations of an object. Each actor acts independently, that is, there is a single thread of control associated with each actor. However the drawback with this model is in what that thread does. Essentially what happens is the thread processes the communications to the actor and it does so in FIFO order. Thus, in a simple actor implementation of a Name Server, all the queries to it would be processes sequentially, regardless of whether there are any access conflicts in the queries.

However there is nothing in the actor model that requires that the processing of a communication be complete before the next behaviour is created. Thus, the Name Server actor can create new behaviours for each query received, effectively creating a team of Name Server actors to do the processing. While this implementation does seem to provide the necessary concurrency, it is also more complex and furthermore, problems arise in trying to handle access conflicts when they do arise since there is no shared state from which to determine the existence of conflicts. The point I am trying to make here is not that an implementation of a Name Server that provides the maximum concurrency is not possible in the actor model, but just that it must be quite complicated, more than seems need be.

6.4 Emerald

Emerald is an object-oriented language developed primarily for the support of building distributed applications [Black *et al.*, 1986, Black *et al.*, 1987, Hutchinson, 1987] but also makes a good general-purpose programming language [Raj *et al.*, 1991]. All entities in an Emerald application are objects. There are two kinds of objects: passive and active objects. A passive object consists of local state and a set of operations. Any other object may invoke an operation to learn about or manipulate the object's state. Invocations are synchronous, such as with a procedure call or Smalltalk-80 message send. An active object also has local state and operations but it also contains an independent *process* that can invoke operations on other objects. When an invocation is made on an object, because invocations are synchronous, the thread of control associated with the process can be thought of as moving to the invoked object. During the execution of an operation, invocations may be made on other objects. As a consequence, the thread of control originating in an active object may span several objects.

Synchronization is provided by monitors. Because Emerald objects completely encapsulate their state, monitors provide a natural way to do synchronization since it can be done locally.

Emerald's concurrent programming model is very close to the one I am proposing. It allows a thread to be associated with an object that, unlike actors, provides the concurrency of the abstraction rather than handling of operation invocations. Thus the Producer/Consumer system can be implemented as two active objects for the Producer and Consumer and a passive object for the buffer. The Producer and Consumer objects have references to the buffer object as part of their state. Since the Producer is an object, it can provide operations to access such state as the number of items produced.

The maximum amount of concurrency in an Emerald application at any time is exactly the number of active objects. Whether the actual concurrency reaches that amount depends on the locking that takes place. Thus, the Name Server can be implemented as a passive object with the appropriate operations (for example, *Lookup* and *Modify*) and the users are all active objects. When a user invokes the *Lookup* operation, what happens is that the thread associated with the user moves to the Name Server object to perform the operation. Thus, assuming the implementation of the Name Server is such that it allows simultaneous *Lookup* operations, there can be as many such operations taking place simultaneously as there are users, that is, as there are active objects. Although the Name Server is a passive object and so has no independent thread of its own, it nevertheless allows concurrent access. Furthermore, since all operations on the Name Server must go through the same object, the information necessary to avoid access conflicts can be easily maintained.

The main lack of the Emerald concurrent programming model is its inability to provide the simultaneous creation of a homogeneous set of threads. Thus, in the computerized whiteboard example, if the Whiteboard object wants to notify a set of users of changes to the whiteboard state, the only way to do it in Emerald is to notify each user sequentially. It is possible to overlap the notifications so that there is some concurrency in this operation, however this complicates the design.

7 Other Issues

In this section I briefly mention issues that I have ignored until now and discuss some of the implications of the model I am proposing. Some of the issues are, I claim, not relevant to the goals of this paper, some are relevant but in ways that are more to do with the implementation of my proposed model, rather than its design, and some I just do not have complete answers to.

7.1 Inheritance

Most people, when confronted with the phrase "object-oriented," immediately think of inheritance. Inheritance, as it is understood in languages such as Smalltalk-80, C++, Trellus,[2] or Eiffel,[3] is a mechanism for *code-reuse*, that is, a way to avoid reimplementing common software modules. As such, the only interaction inheritance, or any other code-sharing mechanism, should have with my concurrency model is that it correctly supports it. Thus I consider inheritance a non-issue.

7.2 Types and Concurrent Programming

There have been many discussions regarding type systems for object-oriented languages. A type system's primary goal is the early detection of errors [Cardelli and Wegner, 1985, Donahue and Demers, 1985]. It does so by providing a means for classifying entities according to their intended use. I consider a system to be *strongly-typed* if no value can be interpreted as another value. Under this definition all object-oriented systems are strongly-typed since no object will respond to invocations to non-existence operations and thus can never be interpreted as something they are not. Since a type system only applies to the interpretation of entities, it should not affect, nor be affected by, the concurrency model provided by the language and so this too is a non-issue.

7.3 Synchronization

I have said very little about synchronization so far. I believe that, in an object-oriented system, each object should look after it's own synchronization. This means that there be some mechanism to control access to the local state of an object.

Recall that the local state of an object is a set of objects and the question arises as to whether the top level object has to worry about access to the local state of the objects that are its local state. If object `A` has object `B` as part of its local state, should `A` worry about the fact that references to `B` may exist outside `A`'s control? One can imagine all kinds of complicated locking rules being applied to ensure there are no such problems. However the solution becomes clear by remembering that an object is a concrete representation of an

[2] Trellus is a trademark of Digital Equipment Corporation.

[3] Eiffel is a trademark of Interactive Software Engineering

abstraction; if it makes sense for the abstraction that `B` corresponds to to be accessed outside of `A` then the implementation of `B` will enforce any access restrictions required, if it does not make sense and there is a reference to `B` outside of `A`, then that's a programming error, not a problem with the synchronization mechanism. Thus an object need only be responsible for access to the set of references that make up its local state.

7.4 Threads as Objects

Since "thread of control" is an abstraction it seems reasonable that a thread be an object as well, that is, have a set of operations that represent a threads behaviour. The kinds of operations a thread might include are *Fork*, *Join*, *Sleep*, *Wakeup*, *Terminate*, and *SetPriority* but would depend on the synchronization and locking mechanisms provided. The problem is determining which object these operations should be invoked on. To treat the thread of an object as a separate object, that is, there is a separate reference to the thread somewhere causes problems as was discussed in section 4; however to be able to apply "thread" operations to any object completely violates the fact that the object is a concrete representation of some abstraction whose behaviour is described by the operations associated with it.

One possibility is to simply not allow such operations to be invoked by an applications programmer (they must exist in some form for the runtime system of the language to be implemented). This solution is unsatisfactory since it would be nice to be able to implement the runtime system for a language in that language. Another possibility is to make the thread reference implicit and use a language construct to get at it. This might be an acceptable solution since it allows the thread management to be ignored by the applications programmer that doesn't care to deal with it. However solving this problem is considered future work.

8 Conclusions

In this paper I have presented a new concurrency model for object-oriented programming languages. The model associates one thread with each object whose purpose is to provide execution of a sequence of actions related to the abstraction the object represents. Concurrency within an object occurs through synchronous invocation of that object's operations by threads belonging to other objects. Invocation may be made of an operation of one object or a set of homogeneous objects. Each object is responsible for controlling access to its own local state and nothing else.

I think the more interesting aspect of this work is how this model was developed. I went back to the first principles of what objects are, namely concrete representation of abstractions, and reasoned that the best concurrency model for such systems must reflect the forms of abstract concurrency that exist. An interesting outcome of this approach is that this model was developed with a object-oriented programming in mind. This is in contrast with claims that concurrency is orthogonal to object-oriented programming. I hope that this approach will promote more useful discussion about concurrency object-oriented programming languages, and programming language design in general.

Acknowledgements

Many of the ideas presented here were inspired by my experience with Emerald and I wish to acknowledge the members of Emerald's design team: Andrew Black, Norman Hutchinson, Eric Jul, and Hank Levy. The current formulation of these ideas are a result of many discussions with Rajendra Raj. Discussions with Jeff Chase, Peter Damron, and Dave Socha also contributed.

References

[Agha and Hewitt, 1987] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 49–74. The MIT Press, Cambridge, Massachusetts. U. S. A, 1987.

[Agha, 1986a] Gul Agha. *A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts. U. S. A, 1986.

[Agha, 1986b] Gul Agha. An overview of actor languages. In *Special Issue of the SIGPLAN Notices on the object-oriented programming workshop*, pages 58–67, October 1986.

[Agha, 1988] Gul Agha. Fundational issues in concurrent computing. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 59–65, September 1988. Published as SIGPLAN Notices 24(4). April, 1989.

[Black *et al.*, 1986] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. Object structure in the Emerald system. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 78–86, October 1986. Published as SIGPLAN Notices 21(11). November, 1986.

[Black *et al.*, 1987] Andrew P. Black, Norman C. Hutchinson, Eric Jul, Henry M. Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[Black, 1985] Andrew P. Black. Supporting distributed applications: Experience with Eden. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 181–193, December 1985.

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

[Donahue and Demers, 1985] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[Hoare, 1985] C. A. R Hoare. *Communicating Sequential Processes.* Prentice Hall, Inc.,, 1985. Section 7.2.1, page 225.

[Hutchinson, 1987] Norman C. Hutchinson. Emerald: An object-based language for distributed programming. Ph. D. thesis. TR 87-01-01, Department of Computer Science and Engineering, University of Washington, Seattle. U. S. A, January 1987.

[Levy and Tempero, 1991] Henry M. Levy and Ewan D. Tempero. Modules, objects and distributed programming: Issues in RPC and remote object invocation. *Software—Practice & Experience*, 21(1):77–90, January 1991.

[Lim and Johnson, 1988] Jeong Lim and Ralph E. Johnson. The heart of object-oriented concurrent programming. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 165–167, sep 1988. Published as SIGPLAN Notices 24(4). April, 1989.

[Raj *et al.*, 1991] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software—Practice & Experience*, 21(1):91–118, January 1991.

[Yokote and Tokoro, 1986] Yasuhiko Yokote and Mario Tokoro. The design and implementation of ConcurrentSmalltalk. In Norman Meyrowitz, editor, *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 331–340, October 1986. Published as SIGPLAN Notices 21(11). November, 1986.

[Yokote and Tokoro, 1987] Yasuhiko Yokote and Mario Tokoro. Concurrent programming in ConcurrentSmalltalk. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. The MIT Press, 1987.