The Application of Concurrent Object-Oriented Techniques to Reactive Systems

Dennis G. Kafura and R. Greg Lavender

TR 92-12

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

April 13, 1992

The Application of Concurrent Object-Oriented Techniques to Reactive Systems

Dr. Dennis Kafura
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061-0106
kafura@cs.vt.edu

R. Greg Lavender
MCC
3500 W. Balcones Center Dr.
Austin, TX 78759-6509
lavender@mcc.com

Abstract

Reactive systems are so named because they are intended to sustain an extended interaction with an environment which evolves outside of the direct control of the program of the reactive system. A language and system model combining concurrency, abstract communication and an object orientation offers several advantages in the design and implementation of large-scale reactive systems. An object-orientation captures the abstraction and variety of entities inhabiting the environment while the autonomy of actual entities is clearly reflected by expressions of concurrency in the program of the reactive system. Abstract communication is necessary to achieve data sharing among heterogeneous systems. However, attempts to design and implement a paradigm unifying these three features have encountered unexpected difficulties. These difficulties include: the interference between concurrency control (synchronization) and inheritance, inadequate application-oriented communication abstractions, the absence of a useful model of exception handling for concurrent object-oriented applications, and the lack of a powerful and useful theory of computation based on asynchrony.

I. Introduction

A distinction may be drawn between *transformational* programs - those intended to terminate after producing a final result, and *reactive* programs - those intended to maintain an ongoing interaction with an external environment which itself evolves outside of the direct control of the program [Manna&Pnueli] [Harel et.al.]. Into this later category fall such familiar programs as operating systems, embedded control systems, command and control systems, and human-computer interfaces. The term reactive *systems,* emphasizes the interactive character of the software - in the simplest case a reactive program interacts with its environment while in more

realistic systems the "program" is a collection of elements interacting with each other as well as with their common environment.

This paper considers what might be termed *large scale* reactive systems. Distinguishing characteristics of such a system are the following. First, the system has considerable *spatial* extent. In different applications the system's components may stretch over metropolitan areas, may involve interaction among components located around the globe or may integrate space-based and ground-based components. Second, the system's components exhibit strongly *autonomous* behavior. The interactions are best viewed as the collaboration of peer entities each of which has a separate purpose and existence Third, the components are both *numerous* and *specialized*. These components are more likely to be considered "agents" than either clients or servers as these later terms are understood presently.

The technical challenges raised by large scale reactive systems are in three related areas: communication, concurrency and representation. The spatial distribution of the system implies the need for communication among heterogeneous machines using possibly different lower layer protocols. The highly autonomous nature of the components implies pervasive concurrency, each agent possessing its own thread of control. This granularity of concurrency is finer than "process" concurrency supported on most commercially-available systems (exceptions exist). Finally, the need to model, design and implement numerous, specialized agents implies a programming paradigm founded on principles of generalization/specialization in design and modeling and encapsulation in implementation.

The technologies relevant to these technical challenges are: OSI upper-layer protocols, actor-based concurrency and object-oriented programming. These technologies are briefly discussed here and in greater detail, as needed, in subsequent sections.

The OSI model and protocols, embodied in international standards, helped to initiate the popular concept of "open computing" among heterogeneous systems. Universal adoption of OSI was prevented by a combination of market and technical forces centered on the lower layer (i.e., transport) protocol. However, it is the upper layer protocols which are increasingly recognized as the area where substantial engineering effort should be focussed [Clark&Tennenhouse]. It is also the upper layer protocols which are relevant to large scale reactive systems. In such systems, the significant communication issues are those related to connection establishment, data translation and synchronization. The

services of the lower layers (routing, flow control, error control) are necessary but exert far less impact than the upper layers on the character of applications. Fortunately, the upper layer OSI protocols can be implemented above different transport protocols [Rose86]. Unfortunately, the current implementation of the upper layer protocols are not suitable because the complexity of the service interface is not effectively abstracted. The solution to this problem is stated below and discussed in Section 3.

Actor-based concurrency [Agha] provides a suitable foundation for designing and implementing agent-based systems. The features of the actor model that are important in this context are: *autonomy* - each actor operates independently with all other actors; *reactivity* - an actor is driven by message directed to it by other actors; *interaction* - the message passing among actors does not diminish the autonomy of the communicating actors and avoids the imposition of unnecessary, complicating control structures; *evolution* - in response to changes in the real world, an executing system of actors can extend (contract) itself dynamically through the creation (destruction) of actors and by changing the topology of communication among existing ones; *encapsulation* - the private state of an actor is secure from unintended manipulation and allows the actor to react in a time-sensitive (history dependent) manner.

Object-oriented programming provides a language and modeling framework within which limited forms of domain knowledge can be represented. Inheritance, the defining feature of object-oriented languages [Wegner90a], allows specialization and variety among entities to be explicitly captured. A class which extends or redefines attributes of an existing class is an example of specialization. Classes derived by specialization from a common class allows the variety of similar, but distinct, entities to be represented. Codifying these relationships allows a substantial model of the real world to be defined. The combination of actor-based concurrency and object-oriented programming allows both the variety and specialization of entities and their autonomy to be represented. This representation is more accurate than one which can be achieved by using either of the technologies separately.

The synthesis of upper layer OSI protocols, actor-based concurrency and object-oriented programming is extremely powerful but it is not without difficulties. A number of the major difficulties are: *synchronization* -the interference between inheritance and concurrency control; *service complexity* - the immaturity of application-oriented abstractions at the service interface of the upper layers of the protocol stack; *exceptions* - the absence of a suitable

model for exception handling; *theory* - the immaturity of algebraic models for asynchronous, dynamic, concurrent objects. The first two of these problems are discussed in Section II and III, respectively, while Section IV discusses the last two problems. Section V is a summary.

The subsequent sections of this paper discuss the problems noted above in the context of three systems:

ACT++: a system for exploring the synthesis of object oriented programming using C++ and actor-based concurrency [Kafura&Lee90];

OOSI: a re-engineering of the upper layer OSI protocols in C++ [Lavender,Kafura&Tomlinson];

Synergy: a framework for distributed, heterogeneous, multi-language applications incorporating both ACT++ and OOSI [Kafura&Lavender].

The first of these systems is operational, the second is being implemented and the third has only begun in recent months.

II. Synchronization

The *observable behavior* of an object is concerned with the set of messages that the object will accept at a given point in time, or alternatively, the set of methods that are visible in the interface of the object upon receipt of a message. The term observable behavior captures the concern with how the object appears to those clients that communicate with the object. Our notion of observable behavior is motivated by the similar notion described in [Milner]; however, in our work to date the machinery of CCS is used in specifying and reasoning about the observable behavior of individual objects, not systems of objects.

In dealing with the observable behavior of *concurrent objects* (also termed concurrent behavior), the relationship between the (internal and invisible) state of an object and the subset of methods which define its (external and visible) observable behavior is critical. This relationship is precisely what defines the semantics of a concurrent object. In order to understand concurrent object behavior, we must investigate this relationship.

The concurrent behavior of an object is captured in part by the class definition of the object and in part by the mechanism employed by the class to guarantee synchronization.

The inheritance anomaly occurs when we attempt to specialize concurrent behavior using an inheritance mechanism. The anomaly occurs because the inheritance mechanism and the synchronization mechanism interfere with one another, limiting the ability of the subclass to reuse the method implementations of the superclass. Furthermore, the anomaly has been observed across a spectrum of concurrent object-oriented languages regardless of the kind of synchronization mechanism employed [America], [Briot&Yonezawa], [Kafura&Lee89], [Nierstrasz].

The types of concurrent object-oriented systems we are interested in are composed of actor-like objects with properties similar to those described in [Agha]. Each object possesses its own thread of control and communicates with other objects via message passing. Concurrency in our system is limited to inter-object concurrency which is achieved using message passing and an actor-like *become* operation. The become operation results in a *replacement behavior* (object) with its own thread of control. Fine-grained intra-object concurrency is not a feature of objects in our system.

We are specifically interested in expressing and inheriting concurrent object behavior in ACT++ [Kafura&Lee90] a prototype object-oriented language based on the Actor model and C++. ACT++ is a collection of classes which implement the abstractions of the Actor model and integrates these abstractions with the encapsulation, inheritance, and strong-typing features of C++. The language falls in the heterogeneous category of concurrent object-oriented languages [Papathomas] since we have both active and passive objects. Active objects are instances of any class derived from a special *Actor* class. Any instance of a class not derived from the Actor class is a passive object. Concurrency is achieved using an actor-like become operation which is implemented in the Actor class. The become operation permits an object to specify a replacement behavior.

There is a general consensus that we do not yet fully understand what it means to inherit concurrent behavior [Wegner90b]. To provide a solid foundation for approaching this issue we have sought a guiding formalism. A formalism based on CCS is presented which exposes the essential elements of concurrent object behavior and leads to conditions which must exist if the inheritance anomaly is to be avoided.

The notion of *behavior abstraction* was previously proposed in ACT++ [Kafura&Lee89] as a mechanism for capturing the behavior of an object. Upon initial examination, behavior abstraction seems powerful since synchronization can be achieved naturally by dynamically modifying the visibility of

the object interface using the become operation. The efficacy of this mechanism and its degree of interaction with the ACT++ inheritance mechanism has been examined by others [Papathomas], [Matsuoka,Wakita&Yonezawa] and has been found to have serious limitations. The most serious limitation occurs because a behavior abstraction is not a first-class entity in the language and is thus subject to the effects of the inheritance anomaly.

Enabled sets [Tomlinson&Singh] improve on the notion of behavior abstraction by promoting the control of the visibility of an object's interface to a dynamic mechanism which can be manipulated within the language; i.e., enabled sets in Rosette are first-class entities.

The flexibility offered by enabled sets caused us to investigate the combination of behavior abstraction and enabled sets which resulted in the notion of a *behavior set* [Lavender&Kafura]. The ACT++ mechanism which captures the idea of a behavior set has the following properties:

- it is a natural extension of formal methods for specifying concurrent object behavior,
- it does not interfere with the ACT++ inheritance mechanism,
- it is free from known inheritance anomalies,
- it can be expressed entirely within ACT++, and
- it can be enforced efficiently at run time.

To represent concurrent object behavior within the ACT++ language, we rely on three first-class entities expressible within the language:

- *state functions* whose boolean results represent tests applied to the current (internal) state of the object,
- a *next behavior function* representing the mapping from internal states (as determined by the state functions) to the observable (external) interface, and
- *behavior sets* representing elements of the observable behavior (powerset of the set of methods).

The excerpt shown in Figure 1 demonstrates how each of these entities is expressed and used in an ACT++ class definition of a bounded buffer. In the example, a class named BoundedBuffer is derived from the base class Actor and defined to have methods named in and out for inserting and

removing items,respectively. The implementation permits at most N items to be maintained at any one time. The state functions, appearing in the protected section of the class definition, are empty() and full() which return true when the buffer contains no elements or can receive no additional elements, respectively. The nextBehavior method uses the state functions to select one of three possible BehaviorSets: Zero - containing only the method in, N - containing only the method out, and Other containing both methods. The initialization of these three BehaviorSets is shown in the class constructor.

```
class BoundedBuffer : Actor {

...//private instance variables

protected:
   BehaviorSet Zero, N, Other;

   virtual bool empty() {...}
   virtual bool full()  {...}

   virtual BehaviorSet nextBehavior() {
        if (empty()) return Zero; else
        if (full() ) return N;    else
                     return Other;
   }

public:

   LinearOrd() {  //constructor
        Zero = BehaviorSet(&in);
        N = BehaviorSet(&out);
        Other = Zero + N;
        become nextBehavior();
   }
}
```

Figure 1: Bounded Buffer expressed with Behavior Sets

In this section we have attempted to explain the relationship between concurrent object behavior and inheritance. In doing so, we are forced to first define the meaning of concurrent object behavior as it occurs in our actor-based concurrent object-oriented language. We have offered a formalized approach for specifying and reasoning about concurrent object behavior based on CCS behavior equations. This approach

emphasizes the relationship between the state of an object and subsets of the set of methods in the interface to the object, called behavior sets. This relationship is embodied in the mapping given by the behavior function. If the inheritance anomaly is to be avoided, behavior sets and the behavior function must be first-class, inheritable, and mutable. We have shown that the language mechanisms of ACT++ (and therefore C++) are sufficiently expressive in this regard.

III. Service Complexity

This section describes features of an object-oriented implementation of the OSI upper layers, called OOSI (ooo-zi), which is in the final stages of development. OOSI is implemented in C++ and is the result of an almost complete re-engineering of the core elements of the ISO Development Environment (ISODE) [Rose89], a widely used research implementation of the upper layer OSI protocols. It is relevant to explore how object-oriented language features can be used in the implementation of a layered protocol architecture. Constructing layered systems using object-oriented techniques is an area of interest to several others as well. Relatively recent work includes Choices [Campbell], the Conduit [Zweig], and OTSO [Koivisto].

The work on OOSI is intended to be of interest to different, audiences. First, we want to communicate to software engineers that the deep integration of object-oriented programing techniques with upper layer protocols facilitates and enhances the implementation of distributed applications. Second, we want to convey to language designers and protocol implementors that object-oriented language features (inheritance, subtyping, and polymorphic functions) are useful in building communication protocols.

A deep integration of object-oriented techniques and layered protocol architectures has profound implications for distributed object-oriented applications. The resulting rich and flexible communications infrastructure is a useful one on which to build next generation distributed applications. In general, development of OSI-based applications, if undertaken at all, is a difficult and time consuming process. Major difficulties are understanding the use of the multi-option service primitives offered by each layer, use of standard application layer service elements such as remote operations, being facile with the Abstract Syntax Notation One (ASN.1) data representation language, and mapping application services onto an appropriate set of communication services.

The difficulties in building OSI-based applications may be overcome by using the boundary surrounding an application entity object to encapsulate:

- complexity: as noted, the existing interfaces to application services entail lengthy argument lists containing system structures which the user must retain and supply with later uses of the service. These system structures can be better represented, and hidden from view, as internal data of objects.

- distribution: an application process may be composed of both locally communicating objects and remotely communicating objects. The application developer need make no, or at least minimal, distinction between objects in these two groups.

- protocol: each pair of peer application entities uses a separate, but not necessarily unique, protocol. The protocol used in interacting with a remote object may be completely hidden from the application developer.

- heterogeneity: the need to encode and decode data passed among systems with dissimilar data representations is a time-consuming and error-prone effort. The encoding and decoding functions can be treated as methods of a class, hiding from the user the details of how the object passes data to remote objects. Furthermore, the existence of a class hierarchy simplifies the programming of the encoding and decoding methods.

Structuring application service elements as objects creates an application environment with simpler, more abstract services and one which is safer as the control of arguments and the proper use of defaults can be insured by the class designer.

Finally on this point, we note the experience of one of the authors with the Carnot project at MCC. Here it is apparent that extensible concurrent object-oriented programming environments, like the one offered by Rosette [Tomlinson,Scheevel&Singh], in conjunction with a rich set of communication abstractions, provides the application layer infrastructure for a much broader open systems technology base than is common in current practice. Typical client-server network applications, such as file transfer, electronic mail, and remote virtual terminal, impose relatively weak demands on the upper layer protocol abstractions. However, of more interest are powerful application models based on peer-to-peer communication, such as distributed workflow coordination and distributed communicating agents. In programming applications in an

inherently concurrent environment, it becomes apparent that design choices embodied in the communication infrastructure impact, in a limiting way, the applications infrastructure.

The particular appropriateness of object-oriented programming to the implementation of communication architectures is the second issue considered in this section. The OOSI experience shows that the the use of object-oriented programming results in a type structure that reflects the architectural model and increases the run-time performance of the protocol machines at each layer. Having worked with and examined in detail the ISODE code, it became obvious that inheritance, subtyping, and polymorphism could be used to explicate the implicit type structure of the layered protocol machines and improve their efficiency. An aesthetic benefit , which is appealing from a software engineering perspective, is that the type structure reflects the service encapsulation and vertical composition of the layered model, making it easier to map the components of the model to corresponding elements in the implementation. At the same time, the type structure enforces the proper run-time access control, enables compact vertical composition of methods (inlining), and produces an encapsulated and localized state structure for each instance of the protocol machine hierarchy. The performance improvement results primarily from a simplified control structure induced by the explicit type structure and selective function inlining.

This experience is important since it contradicts the assertion by some that layered protocol architectures, such as the ISO Reference Model, necessarily suffer in performance because of layering, thus forcing implementors to violate the layering principle in search of efficiency. The result substantiates the experience of others, notably Clark [Clark&Jacobson], that sound network programming methodology leads to efficient implementations that are also understandable.

Much of the advantages found in OOSI result from its use of vertical integration of layers. In vertical integration each layer consists of multiple protocol machine instances, each with an independent state structure vertically related to the state structure of the protocol machines in adjacent layers. An object-oriented approach based on class inheritance between well-defined types naturally represents the vertical structure. Vertical partitioning has been shown to have a positive influence on performance in certain contexts [Hufnagel].

Applications in which each object is allocated its own thread of control can benefit from the vertical integration used in OOSI since each object can have its own instance of a

protocol machine hierarchy. In addition, the protocol machine implementations can be greatly simplified since the vertical partitioning stream lines the bi-directional control flow between service layers and offers opportunities to optimize the layer interfaces based on the code dependency relationship between objects representing each service. Furthermore, it obviates the need for intra-layer multiplexing within the protocol machines, and inter-layer communication is reduced to superclass/subclass method invocations within a single composite object.

In this section we have briefly described our experience with using object-oriented techniques in the implementation of layered communication protocols. The communication infrastructure thus created was also seen as a desirable basis for the construction of distributed applications. The overall conclusion to be drawn from this experience is that language and system designers need to pursue the deeper integration of language structures and communication abstractions.

Section IV: Exception Handling and Theory

A general view of exception handling reveals that there are five distinct roles: the requestor, the customer, the faulting server, the receiver and the handler. The requestor is the entity initiating the request which ultimately leads to the occurrence of the fault. The customer is the entity which is awaiting the result of the request. The occurrence of the fault, of course, prevents this expectation from being satisfied. The faulting server is the entity which is attempting to perform the request but which is unable to complete its responsibility due to the occurrence of a fault. The receiver is the entity which immediately receives the notification that the server has faulted. The receiver is initially assigned the responsibility for identifying and initiating the handler. The handler is the fifth role. The handler is charged with responding to the fault and taking whatever corrective action is possible or necessary.

In sequential languages a number of the five roles are unified. Usually the requestor and the customer are the same. The faulting server and the receiver may also be the same in cases where a server can deal with (at least some) exceptions which it causes. Finally, the handler may be part of the server or the requestor/customer entity. Furthermore, the sequential ordering among these entities allows the exception handling to be implemented as a "stack unwinding" process. Each layer in the stack, corresponding to an intermediary entity between the requestor and the server, may nominate

handlers for specific exceptions. The stack is unwound until a handler is discovered.

In concurrent languages based on asynchrony the exception handling situation is more complicated because no simple relations exist among the five roles. Beyond the simple fact that the five roles may be assumed by five distinct entities the following complications also apply:

- the requestor may no longer exist at the time of the fault,
- the customer may not know the identity of the server,
- the customer may be known to the server, but possibly not to the receiver or the handler,
- there is no clear model of how the receiver is to be identified, and
- there is no clear model of how the receiver identifies the handler.

The first of these complications implies that the information on why the request was made has potentially been lost and is unavailable in the recovery process. The second implies that a blocked customer may remain indefinitely blocked and has no ability to query the server. The last three points simply underscore the lack of a widely accepted model of how the elements of the exception handling environment should be related.

In this section we also consider, very briefly, recent developments in algebraic theories for asnychronous concurrency.

The development of useful theoretical models is of immense concern. Language designers have recognized the benefits that flow from a language design grounded in a deeper and robust mathematical theory. The best example of this is the theory of the lambda calculus and the class of functional programming languages. However, software engineers and those that build tools for software engineering are coming to realize the need for an adequate theory. Software developers working with concurrency have been especially aware of this need. Only with an appropriate theory can tools and analysis methods be created which operate on more than lexical or simple graphical levels. Determining equivalence and demonstrating behavior require a precise semantics and a deep theory.

Many models of concurrent programming have adopted a synchronous model of interaction. This approach requires that both the sender and the receiver have reached a point in their respective behaviors where they wish to communicate. Mature theories of this form are Hoare's CSP and Milner's CCS.

Actor-based concurrency, and reactive system modeling in general, require, however, an ability to reason about asynchronous forms of interaction. The sender should be able to emit a message without concern for whether the receiver is able to accept this message at the same time.

Quite recently new theories based on asynchronous concurrency have begun to appear. We mention two of these while acknowledging that others of which we are unaware may quite likely exist. In 1990 Berry and Boudol [Berry&Boudol] defined a model of asynchronous concurrency based on a paradigm of chemical interactions. Termed the Chemical Abstract Machine, the model envisioned molecules (processes) interacting via reaction rules (messages). The global state of the system defines a chemical solution undergoing stirring by a magical mechanism. Encapsulation was introduced via a membrane. In 1991 Honda and Tokoro [Honda&Tokoro] proposed a theory based on objects communicating via messages in an asynchronous fashion. Their theory includes a notion of bisimulation similar to that in CCS. A number of very small examples hint at the utility of the theory for realistic problems. It is our goal to explore the utility of the Honda and Tokoro theory in more detail.

The relevance temporal logic should also be noted. The formalism of temporal logic, allowing one to reason about the sequences of events in an execution trace, also seems to be clearly relevant to the study of reactive systems. This relevance is reflected in the title of the recent book on this subject [Manna&Pnueli].

Section V: Conclusions

In this paper we have considered a number of issues related to large-scale reactive systems. This term is taken to describe a class of problems in which the dominant features are those of objects, communication and concurrency. The reported experience with ACT++ and OOSI leads to the conclusion that:

- the interference between inheritance and concurrency can be moderated so as to achieve a useful form of interface

control which integrates well with an inheritance mechanism, and

- an object-oriented structuring of layered communication protocols is beneficial to both the form and performance of the protocol software and is also a proper foundation for the construction of distributed applications.

A number of major issue remain unsettled. Two of these issues are the definition of a model for exception handling and the development of a pertinent theory for asynchronous concurrency. Promising work in the later area was noted.

References

[Agha] Gul Agha. Actors: A Model of Concurrent Computation in Distributed Systems, M.I.T. Press, 1986.

[America] Pierre America. "Inheritance and subtyping in a parallel object-oriented language," *ECOOP'87 Proceedings*,pp. 234-242, Springer-Verlag, 1987.

[Berry&Boudol] Gerard Berry and Gerard Boudol, "The chemical abstract machine," conference Record of the Seventeeth Annual ACM Symposium on Principles of Programming Languages, San Francisco, CA January 17-19, 1990, pp. 81-94.

[Briot&Yonezawa] Jean-Pierre Briot and Akinori Yonezawa., "Inheritance and synchronization in object-oriented concurrent programming,"in ABCL: *An Object-Oriented Concurrent System*, (ed. A. Yonezawa), MIT Press, 1990.

[Campbell] Roy H. Campbell, Gary M. Johnston and Vincent F. Russo. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)," Operating Systems Review, No. 21, July 1987, pp. 9-17.

[Clark&Jacobson] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. "An analysis of TCP processing overhead," IEEE Communications Magazine, June 1989, pp. 23-29.

[Clark&Tennenhouse] David D. Clark and David L. Tennenhouse. "Architectural considerations for a new generation of protocols," ACM SIGCOMM'90, pp. 200-208.

[Harel et.al.] "STATEMATE: A working environment of the development of complex reactive systems," IEEE Transactions on Software Engineering, Vol. 16, No. 4, April 1990, pp. 403-414.

[Honda&Tokoro] Kohei Honda and Mario Tokoro, "An object calculus for asynchronous communication," Proceedings, ECOOP '91, pp. 133-147.

[Hufnagel&Browne] Stephen P. Hufnagel and James C. Browne. "Performance properties of vertically partitioned object-oriented systems," IEEE Transactions on Software Engineering, 15(8),August 1989, pp. 935-946.

[Kafura&Lavender] "The synergy between object-oriented programming and open system interconnection," Workshop on Harnessing in the Object-Oriented Revolution, Denver Colorodo, February, 1992.

[Kafura&Lee89] Dennis G. Kafura and Keung Hae Lee. "Inheritance in actor-based concurrent object-oriented languages," ECOOP'89 Conference Proceedings, Cambridge University Press, 1989, pp. 131--145.

[Kafura&Lee90] Dennis Kafura and Keung Hae Lee. "ACT++: building a concurrent C++ with actors," *Journal of Object-Oriented Programing*, Vol. 3, No. 1, pp. 25-37, May/June 1990.

[Koivisto 1990] Juha Koivisto and Juhani Malka. "OTSO - an object-oriented approach to distributed computation," Usenix C++ Conference Proceedings, April 1991, pp. 163-177.

[Lavender&Kafura] "Specifying and inheriting concurrent behavior in an actor-based object-oriented language," TR 90-56, Department of Computer Science, Virginia Tech, Blacksburg, VA 24061-0106.

[Lavender,Kafura&Tomlinson] "Implementing Communication Protocls Using Object-Oriented Techniques," submitted to OOPSLA '92.

[Manna&Pnueli] *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.

[Matsuoka,Wakita&Yonezawa] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. "Analysis of inheritance anomaly in concurrent object-oriented languages," extended abstract presented at the ECOOP/OOPSLA'90 Workshop on Object-based Concurrency, October 1990, to appear in SIGPLAN Notices.

[Milner] Robin Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

[Nierstrasz] Oscar Nierstrasz. "Active objects in hybrid," *OOPSLA'87 Proceedings*, pp. 243-253, 1987.

[Papathomas] M. Papathomas. "Concurrency issues in object-oriented languages," in *Object Oriented Development*, pp. 207-245, (ed. D. Tsichritzis), Centre Universitaire D'Informatique, Universite De Geneva, 1989.

[Rose86] Marshall T. Rose. "OSI transport services on top of the TCP," *Computer Networks and ISDN Systems*, 12(3), 1986.

[Rose89] Marshall T. Rose. The Open Book: A Practical Perspective on OSI, Prentice-Hall, 1989.

[Tomlinson&Singh] Chris Tomlinson and Vineet Singh. "Inheritance and synchronization with enabled-sets," ACM OOPSLA'89 Conference Proceedings, October 1989, pp. 103-112.

[Tomlinson,Scheevel&Singh] Chris Tomlinson, Mark Scheevel, and Vineet Singh. Report on Rosette 1.1, MCC Technical Report ACT-OODS-275-91, July 1991.

[Wegner90a] Peter Wegner. "Concepts and paradigms of object-oriented programming," *OOPS Messenger*, Vol. 1, No. 1, pp. 7-87, August 1990.

[Wegner90b] Peter Wegner. Discussion Panel on Issues in Object-based Concurrency, held in conjunction with ECOOP/OOPSLA'90, October, 1990.

[Zweig] Jonathan M. Zweig and Ralph E. Johnson. "The conduit: a communication abstraction in C++," 1990 Usenix C++ Conference Proceedings, April 1990, pp. 191-203.