

Issues in Concurrent and Distributed  
Objects  
Kevin L. Mills

INFT 821 Fall 1992  
OBJECT-ORIENTED SOFTWARE  
DEVELOPMENT  
George Mason University

# Issues In Concurrent and Distributed Objects

Kevin L. Mills

December 8, 1992

## I. Introduction

Over the past decade object-oriented technology emerged as a force for improved software quality and productivity. During that same decade, continued improvement in processor and memory price-performance increased the availability of computers, encouraged the deployment of networks, and enabled the construction of massively-parallel, compute engines. Unfortunately, most object-oriented technology that has reached the market assumes that objects will be linked into sequential programs, sharing the same address space. This mismatch between currently available object-oriented software techniques and a growing base of multiprocessors and computer networks has created an interesting area of research: concurrent and distributed object-oriented approaches.

This paper: 1) surveys a number of approaches taken by researchers, 2) proposes a taxonomy of issues in concurrent and distributed objects, and 3) examines one specific issue in some depth. The specific issue examined is the utility of abstract data types, as represented by Eiffel classes, to describe an abstract specification for a distributed, peer-to-peer, communications service (the Open Systems Interconnection (OSI), connection-oriented, transport service<sup>33</sup>).

The earliest work on concurrent objects developed a class of approaches called actor models. The two most prominent

examples are Actors<sup>1</sup> and An object-Based Concurrent computational Model 1 (ABCM/1).<sup>70</sup> These models, evolving from work on artificial intelligence, define significant computational elements as objects that have their own thread of control (there are exceptions for some fundamental objects). The primary target for these approaches is highly parallel, multiprocessors. A second, distinct class of approaches could be called distributed models.<sup>28,47,59,60</sup> These models assume that objects are deployed in loosely coupled networks, possibly with a wide geographic distribution. These models are interesting because distribution introduces a second level of concerns, piled atop the usual problems of concurrency. A third class of approaches can be named extensions to existing object-oriented languages.<sup>27,31,52</sup> These approaches are worth considering because they start with an existing model of sequential, object-oriented programming and then extend that model to include concurrency features. A fourth set of approaches can be grouped together under a category labelled other.<sup>8,13,25,39,55</sup> These miscellaneous approaches are worth considering because each provides a set of features aimed at specific issues that are not addressed in other approaches.

The second part of this paper proposes a taxonomy of issues in concurrent and distributed objects. The initial split of the taxonomy separates concurrency issues from distribution issues. Distribution implies concurrency; thus, having considered the problems that arise when objects execute concurrently, introducing distribution imposes an additional set of difficulties. Guidance is available in the literature on general issues of concurrency<sup>7</sup> and some researchers have proposed a taxonomy of issues for distributed computer systems,<sup>49</sup> but we attempt to provide a comprehensive discussion of both concurrency and distribution issues in an object-oriented context. The major concurrency topics considered include:

granularity of parallelism; method of communication; method of synchronization; approach to atomicity; the object life model; the knowledge sharing model; and the exception model. The major distribution issues discussed include: model of cooperation; model of migration; method of encapsulation; means of naming, addressing, and locating; heterogeneity; replication; and security.

The third part of this paper presents a case study on a specific issue: the utility of Eiffel classes for specifying an abstract service across distributed objects that provide a peer-to-peer, connection-oriented, data transport service. Two approaches are presented, analyzed, and evaluated. The first approach maps the natural language OSI transport service specification<sup>33</sup> onto abstract data types (ADTs) represented as Eiffel deferred classes. This requires establishing a model for interaction between users of the transport service and a service provider. The service provider, in an implementation, consists of objects that are distributed between two nodes in a network. This creates some interesting issues when applying Eiffel pre- and post-assertions to specify the service provided to users.

The second approach rearranges the model into a more realistic *programmer's view* that attempts to describe the abstract service interface as an application programming interface. The rearrangement is accomplished without introducing semantic changes into the OSI transport service. The purpose of this second approach is to determine if the OSI transport service can be represented as an application programming interface (API) specified in Eiffel. Both the ADT and API approaches are evaluated, and a general evaluation is also presented regarding the utility of Eiffel assertions for specifying abstract service interfaces to distributed, peer-to-peer, communications protocols.

The paper closes with some general conclusions about issues of concurrency and distribution in object-oriented systems. A list of references and related papers follows the concluding section. Two appendices are provided. Appendix A contains an Eiffel Abstract Specification of IS 8072: The Connection-oriented Transport Service. Appendix B contains an Eiffel Abstract Specification of an Adapted, Programmer's Model of the Connection-oriented Transport Service.

## II. Concurrent and Distributed Object Models

A growing trend toward multiprocessing and computer network deployment has stimulated research on issues of concurrency and distribution. In fact, object-oriented paradigms, traditionally synchronous and sequential in nature, must be updated to account for requirements of concurrent operation and geographic distribution. Although no consensus exists on the best means to incorporate concurrency and distribution into object-oriented paradigms, researchers have proposed a number of models. In this section, we survey some of the proposed models. Our survey is organized into four categories: 1) actor models, 2) distributed models, 3) object-oriented language extensions, and 4) other models.

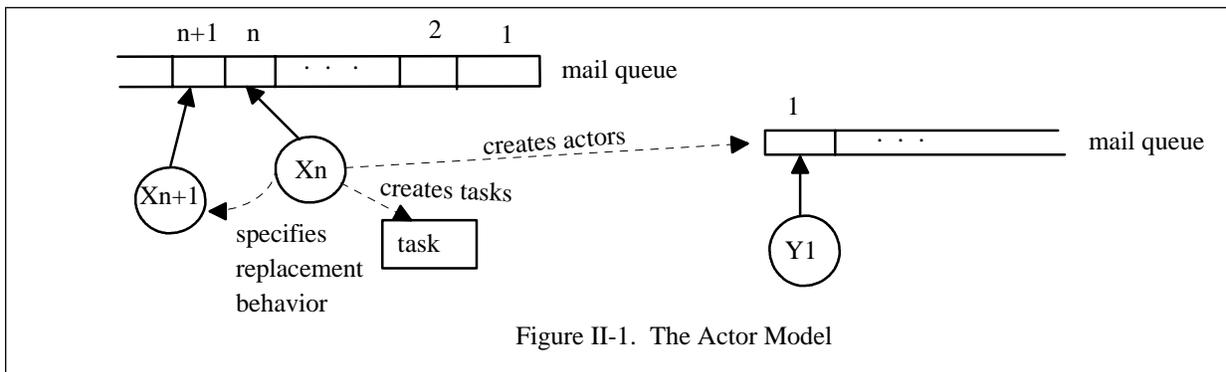
### A. Actor Models

The original actor model was proposed by Hewitt<sup>3</sup>, formalized by Agha<sup>1,5</sup>, and implemented as a series of actor languages.<sup>2,36,44</sup> A second actor model, An object-Based Concurrent computational Model 1 (ABCM/1), was developed by Yonezawa<sup>70</sup> and implemented as An object-Based Concurrent Language 1 (ABCL/1).<sup>65,68,69</sup> We describe each of these models, and the associated languages, in turn.

*Actors.* The Actors Model of distributed computation, as formalized by Agha, aims to define the minimal set of concepts necessary for distributed computation in massively concurrent architectures. An Actor encompasses an independently executing thread of control that processes one incoming message by performing one or more of three operations: 1) sending messages to other Actors, 2) creating a replacement behavior to process the next message received by the Actor, and 3) creating additional Actors. The behavior of an Actor can be history sensitive; the actions taken by an Actor cannot be presumed sequential. Actor creation is a primary part of the computational model because computations are made increasingly concurrent by assigning parts of a problem to individual Actors.

All communications between Actors use asynchronous, buffered message passing. This uniform communication model permits an Actor to send messages to itself without fear of deadlock. The arrival order of messages into an Actor's queue is assumed to be arbitrary, so messages arriving at a queue simultaneously are placed into the queue in fair, but unpredictable order. In addition, the Actor Model assumes that all messages sent will be delivered eventually.

Computation among a system of Actors is performed in response to messages, synonymous with tasks, sent to the system. Each task consists of three components: 1) a tag that uniquely identifies the task within the Actor system, 2) a target which



is the mail address to which the message is to be delivered, and 3) a message identifying the operation to be performed by the target Actor and specifying any parameters needed. To send a task to an Actor, the mail address of the target must be known by the sender. As computation proceeds, new Actors are created within the system, as well as new tasks. As tasks are completed and Actors are no longer needed, garbage collection removes them.

Each Actor is described by specifying a behavior and a mail address for the Actor. The Actor Model is illustrated in Figure II-1.

An Actor,  $X_n$ , is shown accepting a message,  $n$ , creating a task for another Actor (not shown in the figure but already known to  $X_n$ ), creating another Actor,  $Y_1$ , and specifying a replacement behavior for itself,  $X_{n+1}$ , to process the next message,  $n+1$ . All control in the Actor Model results from message passing.

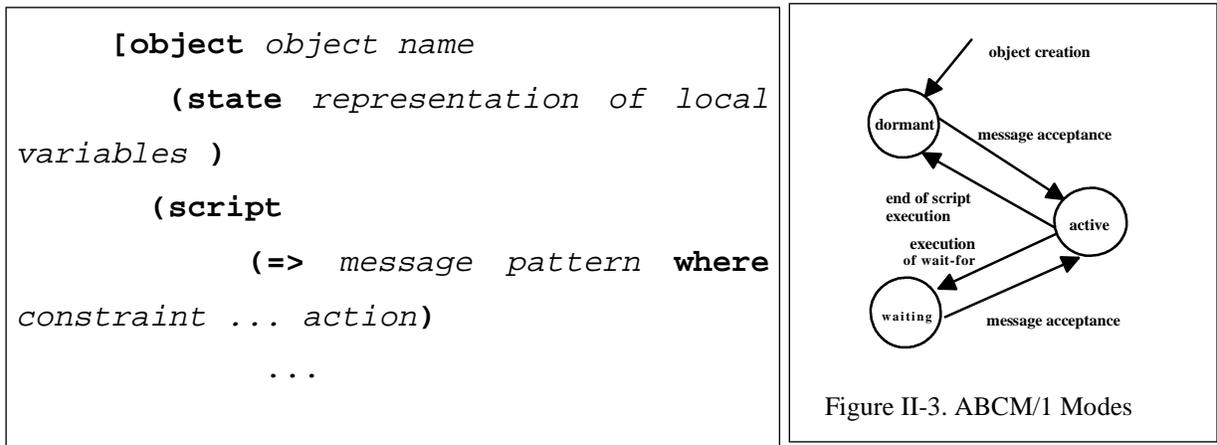
To implement the Actor Model, Agha defined two languages: a procedural language, called a Simple Actor Language (SAL), and a message-passing language, named Act. SAL is intended as a teaching tool, while Act provides a kernel, or minimal, actor language upon which richer languages, such as Act3, can be built. In both SAL and Act, the same facilities are provided.

The most fundamental semantic in an actor language attaches an identifier to a behavior definition. Within a behavior definition, is an acquaintance list (the mail addresses of other Actors known to this behavior) and a list of messages that the Actor can process. Incoming messages are bound to a specific set of operations according to these lists.

A minimal actor language also includes primitives to create new actors and tasks (i.e., send messages) and to identify internal actors, called receptionists, to be known outside of the actor system, and to declare actors, called external actors, that are known from outside of the actor system.

The final set of constructs needed in a minimal actor language include a become command, to specify a replacement behavior, and some conditional statements (if-then-else and case forms). Whenever an actor fails to specify a replacement behavior, the default replacement behavior is identical to that of the actor.

Within a minimal actor language, there are three types of actors: serialized, immutable, and built-in. Serialized actors are sensitive to history and, thus, specification of a replacement behavior must be delayed until the relevant state changes have been made within the actor. Immutable actors are stateless and, thus, need not specify a replacement behavior. Built-in actors can perform their behavior without passing any messages and, thus, prevent the message passing between actors from becoming infinite. More powerful constructs (for variable



function signatures, for delegation, for sequential composition, and for delayed and eager evaluation), not included in a minimal actor language, may be built from the minimal language and then presented in more abstract form as part of a higher level actor language.

In summary, the Actors Model, and related actor kernel languages, define a minimal set of operations that enable a diverse set of computations to be carried out concurrently. Since each function is an actor, the processing overhead in

actor systems can be quite large, unless a method exists to provide extremely light-weight processing and fast message passing. The assumptions about guaranteed delivery of messages could place an impossible burden on a communications system, especially if the individual actors were distributed in a large communications network. The Actors Model is theoretically interesting, but the practicality of actors as an implementation method for distributed systems is questionable.

*ABCM/1 and ABCL/1.* The ABCM/1 model encompasses the concept of objects and the interactions between them. Each object possesses its own thread of control (i.e., is an active object or actor), encapsulates a set of methods that can be invoked by arriving messages, and contains, optionally, local variables that persist across invocations of the object's methods. Each method within an ABCM/1 object comprises a set of operations that may include any of the following: 1) send a message to another object, 2) create another object, and 3) access and alter local variables. The methods that can be invoked within an object, at a given point in time, depend upon the messages that an object can accept. In turn, the messages that an object can accept depend upon the message patterns, the parameter values within a message, and the current state of the object. Figure II-2 gives a simple view of an ABCM/1 object.

An ABCM/1 object is always in one of three states: 1) dormant (awaiting a message), 2) active (a message has arrived that matches a message pattern, including any constraints, in the object), or 3) waiting (the object is waiting on a reply from another object before continuing). If an arriving message satisfies more than one message pattern/constraint pair, then the first pair (from the beginning of the script) that is matched will be invoked, i.e., there is no nondeterminism. Objects with local memory are serialized, i.e., only one message can be processed at a time.

Each object is assumed to own an infinite input queue, but the management rules for that queue are rather unusual. For example, a dormant object examines the queue for a message that matches a pattern-constraint pair within the object. The first such message is processed and all messages that were in the queue ahead of the accepted message are discarded. On the other hand, when the object is in waiting mode, the first message matching any of the awaited pattern-constraint pairs that arrives in the queue is received immediately, but no messages are discarded.

The message passing model within ABCM/1 plays a significant semantic role. All messages must identify one or more receivers (there is no broadcast); therefore, an object must know about other objects in order to send a message. An object may be created knowing about another set of objects, and may also come to know about and forget about other objects during the course of its life (but an object always knows about itself). A message can be sent by an object at any time. Messages are guaranteed to arrive within a finite time and will be buffered upon reception until an object is prepared to process them. The incoming messages are queued in the order in which they arrive. Any messages sent from an object A to an object B are guaranteed to arrive in the same order in which they were sent. (This is, of course, an assumption that cannot be met by most computer networks without appropriate protocols. If these protocols are not provided in the network, then the ABCM/1 model would become very cumbersome.)

Messages sent between ABCM/1 objects are of three types: 1) **past**, 2) **now**, and 3) **future**. **Past** messages are simply asynchronous datagrams, the sender can continue processing after sending a **past** message. **Now** messages are synchronous procedure calls, the sender must wait for a reply after sending a **now** message. (For obvious reasons, an object cannot send a **now**

message to itself.) **Future** messages are asynchronous datagrams that require a reply sometime in the future, but the sender of a **future** message can continue processing. The reply to a **future** message is stored within a **future** variable until the sender of the **future** message is ready to access the reply. If the sender of the **future** message attempts to access the reply before it has been produced, then the sender must wait for the reply.

An interesting and useful addition, over and above the actor model defined by Hewitt, provided by ABCM/1 is the concept of **express** mode and **express** messages. Messages that are sent in **express** mode will interrupt the processing of the receiving object. (Only one level of priority is supported, i.e., interrupt processing cannot be interrupted.) ABCM/1 provides rules that specify when interrupts can be honored. In general, interrupts are not accepted while an object is accessing local variables, nor when an object is executing an atomic block (a sequence of actions designated as atomic by the programmer). Further, the programmer can specify whether normal processing will be aborted or resumed after an interrupt is received. Each type of message may be sent as either a normal or **express** message, leading to the following combinations.

	NORMAL	EXPRESS
Past	[T <= M]	[T <<= M]
Now	[T <== M]	[T <<== M]
Future	[T <= M \$ x]	[T <<= M \$ x]

These message types are expressed in the syntax of ABCL/1. In each case, a message, M, is sent to an object, T. An **express** message is designated by a double arrow, <<. A **past** message is designated by a single equal sign, =. A **now** message is

designated by a double equal sign, ==. A **future** message is designated by a bound (with the \$) variable, x.

The ABCL/1 language that implements the ABCM/1 model does not attempt to represent every concept as an object (as, for example, Smalltalk does), but includes constructs from Lisp. ABCL/1 does, however, embody some key features for parallelism and synchronization. ABCL/1 objects operate concurrently, so, for example, if an object sends multiple messages to different receiving objects, the processing among the receivers will overlap in time. ABCL/1 also permits an object to send multiple messages in parallel and to multicast messages simultaneously to a group of objects. Regarding synchronization, ABCL/1 provides a number of mechanisms: 1) serial execution of actions related to a received message, 2) a wait-for mode that requires an object to suspend processing until an acceptable message is received, and 3) **now** and **future** messages.

In summary, ABCM/1 and ABCL/1 provide a practical refinement to the actor model of Hewitt and Agha. ABCM/1 provides a rich semantics for message passing, synchronization, and parallelism. The addition of **express** mode messages, and the associated interrupt processing capabilities, is extremely useful in real systems. The inclusion of provisions for atomicity yields a programmer-controlled mutual exclusion mechanism. The addition of language constructs from a procedural programming language enable active objects to be reserved for significant concepts within an application, although the simplicity and elegance of the actor model is lost.

## *B. Distributed Models*

Distributed models assume that objects, each encapsulating some significant service, are distributed around a loosely-coupled network of computing nodes. Sometimes, the

nodes are assumed to be heterogeneous, sometimes homogeneous. We survey four particular models: 1) ARGUS, 2) the Common Object Request Broker Architecture (CORBA), 3) the Manager Model, and 4) the Chorus Object-Oriented Layer (COOL).

*ARGUS.* ARGUS is an integrated programming language and system model proposed by Liskov when she was at MIT.<sup>46,47</sup> The main focus of ARGUS is provision of reliable, distributed transactions within a network of computation servers. The assumptions made within the ARGUS model are: 1) the network and the connected computer nodes may be unreliable, but all failures can be detected and 2) the time required to send messages between nodes is long, relative to the time needed to access local memory within a node. ARGUS was designed to meet the following requirements: 1) enable a system to provide reliable, continuous service in the face of node and network failures, 2) facilitate logical and physical changes dynamically while the system continues to operate correctly, 3) permit each node to be managed autonomously of every other node in the system, 4) allow the programmer to control the allocation of modules to nodes, 5) increase processing performance through concurrency, and 6) maintain consistency among the distributed data.

To accommodate these requirements, ARGUS models activities as distributed transactions that are atomic and recoverable. Each transaction will complete totally or will be aborted totally (i.e., the system state will remain consistent). Each transaction is guaranteed to appear to be serial with regard to other transactions.

Nodes in ARGUS communicate through messages, each of which comprise a paired send and reply. The ARGUS model provides only a remote procedure call, with at-most-once semantics (each message is received and acted upon once, or never received, but the sender is apprised of non-receipt.) Liskov argues that this model of message passing allows the communication system to mask

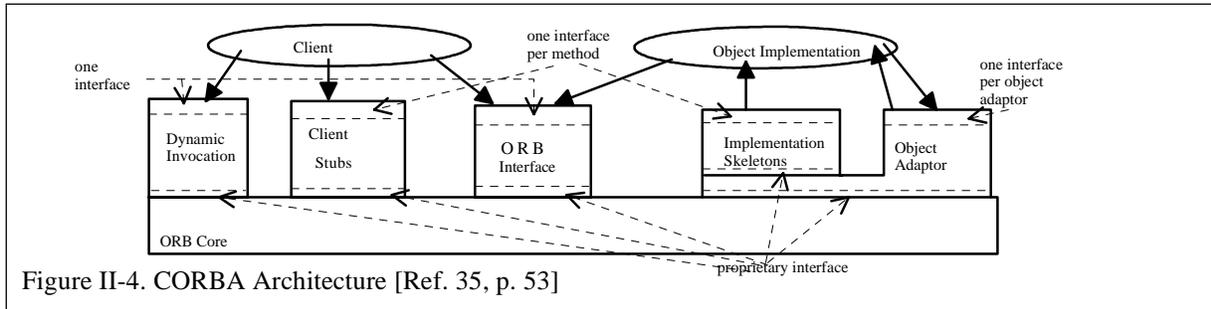
all the details of protocol processing from the application programmer, while giving the programmer exactly the guarantees and information needed to cope with failures. Of course, stop and wait approaches can lead to deadlocks, but ARGUS ignores such possibilities, assuming that a human user will abort a given transaction should the waiting time become excessive.

The architecture of ARGUS relies on a distributed set of guardians. A guardian, encapsulating processes and data, controls access to some resources by checking access rights and by synchronizing concurrent accesses. Processes within a guardian can communicate through shared data, while processes in different guardians can communicate only via messages. Guardians maintain two copies of their data: one in memory and one on secondary storage. The secondary storage shadows the guardian's primary memory, allowing the guardian to resume from the point of interruption after a node failure.

In summary, ARGUS supports distributed services by encapsulating processes and data, by synchronizing access to the processes and data, and by recovering from the point of interruption after a node, on which guardian processes execute, has failed. Multiple guardians can cooperate via synchronous message passing to accomplish application objectives.

*Common Object Request Broker Architecture (CORBA)*. The CORBA defines a framework for linking together client and server objects distributed around a computer network.<sup>9,35,59</sup> A fundamental assumption of CORBA is that the client objects may be developed independently by various companies, may exhibit different interfaces, and may be implemented in an assortment of programming languages. To accommodate these assumptions, CORBA comprises a number of components, some are illustrated in Figure II-4.

The CORBA components shown in Figure II-4 indicate how client and server object implementations are integrated. The



use of both client stubs and dynamic invocations reflects a lack of consensus in the vendor community. (CORBA is a standard architecture being developed on behalf of the computer industry by the Object Management Group (OMG).) One point of view (the static approach) requires that a remote procedure call (RPC) stub be implemented for each method supported by an object. Invocations of the methods then go to the client stubs, from there to the object request broker, or ORB, (the ORB is a component of the ORB Core), onto the appropriate object implementation skeleton and object adapter, where the method is performed. The competing point of view (the dynamic approach) permits calls on methods to have a varied signature. The initial run-time call goes to the dynamic invocation library where the invariant parts of a call are linked to the run-time parameters, from there the path converges with the static approach.

CORBA, as it stands today, does not provide a complete environment for distributed object management (DOM); however, a complete architecture is being specified. In a DOM, every element in a distributed system (e.g., applications, databases, files, batch scripts, and protocols) is modeled as an object. This modeling is accomplished by describing the interface to each object. Objects can then pass messages among themselves to request and fulfill services. The CORBA, to be a DOM, must provide means to: find a satisfactory object, compose service request messages, and bind methods to objects. To accomplish these objectives, CORBA calls for eight components:

- ♦ an object model,
- ♦ the ORB Core,
- ♦ an interface definition language (IDL),
- ♦ a dynamic invocation interface,
- ♦ a static invocation interface,
- ♦ an interface repository,
- ♦ an object adapter, and
- ♦ object implementations.

Some of these components have already been explained, but others deserve discussion.

The CORBA object model is a conventional object-oriented paradigm that includes: objects, requests (or messages or method invocations), types, an object interface, operations, and attributes. Perhaps the CORBA object model is better viewed through the lens of prohibited concepts. CORBA does not allow object aggregation or inheritance, does not define how objects can be linked at run-time, does not address creating, copying, and managing objects, does not include exception handling, and does not enforce object operations as atomic transactions. (These are some major shortcomings.)

The ORB Core comprises a proprietary implementation of functions for object location, message delivery, and method binding. The proprietary nature of the ORB Core is a major deficiency of CORBA as a standard. CORBA allows software vendors to create clients that can be moved from one vendor's environment to another's, but the request brokers of different vendors cannot necessarily work together in a heterogeneous network of ORBs that can find and invoke object operations throughout a network.

The interface description language allows an existing application interface to be described in terms that can be used to compile a mapping between CORBA interfaces and the

application interface. This permits existing applications to be integrated into a network of ORBs without being rewritten.

In summary, CORBA defines an architecture for portability of client objects among server objects implemented by a variety of vendors. While CORBA is extensible, its failure to address some key features of the object-oriented paradigm, coupled with the current proprietary nature of the ORB Core leave many issues open and requirements unsatisfied.

*The Manager Model.* The Manager Model distributes a set of Managers across a network of computing nodes.<sup>60,61</sup> These Managers can cooperate to perform functions through transparent operations (TOP). A TOP allows a Manager to invoke an operation (via a request) on and receive a result (via a reply) from another Manager in a universe of interconnected, heterogeneous nodes.

The Manager Model guarantees three desirable transparencies: addressing, distance, and data representation. Addressing transparency is provided through a uniform, logical name space, coupled to a name service that resolves the specific address of each Manager. Distance transparency is provided by a mapping function that invokes the appropriate communication mechanism depending on where the destination Manager is located. Data representation transparency is provided by defining interfaces using Abstract Syntax Notation 1 (ASN.1) and by using appropriate encoders and decoders for each ASN.1 interface defined.

When a Manager cannot complete a request, the operation may be delegated to another Manager. Managers may be related through creation or through service, the service relationship has two forms: client-server and superior-subordinate (the superior-subordinate relationship generally follows the Manager creation tree).

Three forms of cooperation are incorporated in the Manager Model. Client-server cooperation allows one Manager to invoke services from another Manager. Peer-to-peer cooperation enables two Managers to invoke services from each other to support an application. Cooperative processing involves a Manager that is conducting operations with multiple Managers simultaneously to support an application.

While a Manager appears to be a single entity when viewed from the outside, internally a Manager is composed of a collection of Workers. Workers, the unit of concurrency in the Manager Model, provide the active threads of control for a Manager. Thus, when a request is received, a Coordinator Worker can create a Server Worker and assign the request. The internal behavior of each Worker is specified via a structured finite state machine (SFSM). SFSMs permit behavior to be specified through a hierarchy of FSMs. Some Workers may comprise a single state (i.e., are stateless) and, thus, accept any operation at any time.

Each Worker in a distributed system of Managers is an instance of a Worker Class. The model supports multiple inheritance. Operations between Workers can be invoked synchronously or asynchronously.

In summary, the Manager Model goes further than the existing CORBA to define and implement a distributed, object-oriented model. The flexibility of the Manager Model is greater than that of ARGUS. The Manager Model accommodates heterogeneity by relying on industry standard protocols to provide communications. The Manager Model insulates the application program from issues of communication diversity and computing environment variability.

*Chorus Object-Oriented Layer (COOL)*. COOL was implemented to evaluate the feasibility of managing objects via kernel functions atop which multiple object models can be mapped.<sup>28</sup>

Chorus is a distributed operating system nucleus that permits the inclusion of subsystems. Subsystems run within the nucleus address space and are accessed through system calls.

One Chorus nucleus runs on each network node. The node nucleus manages the operation of actors; actors are analogous to processes in UNIX, except that each actor can support multiple threads of control. Threads within an actor can communicate using shared memory, but communication between actors requires message passing. Chorus supports both synchronous and asynchronous message passing. Messages are passed, transparent to the location of the actors, across ports between actors. Chorus also supports multicasting message passing and a global space of unique identifiers.

COOL is layered on top of Chorus as a subsystem in the nucleus. COOL embodies a set of object managers, one per node, that handle object creation, copying, deletion, communication, and migration. COOL provides an object name server that integrates into the UNIX file naming hierarchy. Objects may be declared as globally known or locally known. The intent of COOL is to support the mapping of distribution and concurrency functions onto existing object-oriented languages, such as C++.

In summary, COOL lays an object management model atop a distributed operating system kernel (Chorus) to facilitate mapping concurrent and distributed object functions onto existing object-oriented languages. COOL assumes a homogeneous network. COOL does not provide an object location facility. The COOL model breaks down as the granularity of the objects decreases.

### *C. Object-Oriented Language Extensions*

Some proposals for concurrency among objects start with an existing object-oriented language and introduce constructs,

including syntax and semantics, that extend the language to provide active objects and synchronization mechanisms. Here we survey three such proposals. One proposal specifies additions, previously developed for C, to C++ that introduce concurrency functions. A second proposal describes mechanisms for supporting real-time requirements by adding language features to C++. The final proposal adds concurrency to Eiffel.

*Concurrent C++.* Two orthogonal extensions to the C programming language were developed at Bell Labs during the second half of the 1980's: one extension, C++, incorporated object-oriented features into C, the other extension, Concurrent C, incorporated concurrency features into C. Researchers at Bell Labs are now combining these extensions to form Concurrent C++.<sup>27</sup>

Concurrent C introduces the concept of processes that can execute in parallel and that can communicate between each other using synchronous and asynchronous message passing. Concurrent C++ adds: 1) potential to encapsulate interfaces to processes within classes and to represent messages as classes and 2) invocation of constructors and destructors at process creation and termination. Objects that are shared by multiple processes can be encapsulated in a guard process.

At the time the referenced report was written a number of integration issues, mainly relating to keeping the flavor of C++ while adding concurrency features, were unresolved. For example, passing references to objects between processes is not possible, so a method of object reference between processes must be devised. As another example, although deriving one process from another using C++ inheritance is desirable, unfortunately this seems to be impossible. In fact, merging classes and processes into a unified concept, a natural approach in many concurrent object languages, was still being investigated for C++. As a final example, object methods in C++ can be

overloaded, but no decision had yet been made about overloading messages, the concurrent counterpart to method invocations, passed between processes.

In summary, the effort at Bell Labs to combine concurrency into C++ illustrates a drawback common to all language extension approaches: new features must be added in a manner that preserves the existing semantic model of the language. This requirement limits technical options for adding concurrency.

*RTC++*. *RTC++* extends C++ to include: 1) active objects, 2) optional specification of timing constraints on methods and even individual statements, and 3) optional specification of periodic tasks with hard timing. *RTC++* also includes language mechanisms to avoid priority inversion and to allow inheritance among active objects. *RTC++* allows concurrent execution among the methods in an active object, but each method may, itself, only be executed serially, i.e., by one thread at a time.

Active objects in *RTC++* are declared as active classes. By default, an active object has one thread of control, but multiple threads of control can be specified. Threads may be slave threads, invoked by a method request, or master threads, executing independently in the background. When a slave thread is invoked, the slave inherits the priority of the caller; this avoids the priority inversion problem. If so specified, slave threads may be interrupted when a method is invoked by a higher priority thread; however, if the interrupted thread is executing in a critical region, the interrupted thread will continue execution until it exits the critical region.

*RTC++* introduces a guard expression that can delay execution of a method until a specific condition is met. A guard expression may include specification of a function to be executed if the condition is not met. Guard expressions are also used to implement critical regions.

All message communication in RTC++ is synchronous. Replies can be sent by two mechanisms: 1) return statement (the reply is sent and the remote procedure terminates) or 2) reply statement (the reply is sent and the remote process continues execution).

RTC++ provides mechanisms to catch and handle exceptions from an object, from a thread, or from a kernel. Language constructs also allow protected regions to be declared; exceptions are not permitted while a protected region is being executed.

RTC++ includes a number of timing facilities. Each RTC++ method may be augmented with a deadline and a specification of the function to invoke if the deadline is not met. Individual RTC++ statements may be augmented with timing constraints: execute within, at, or before a specified time. For master threads, timing cycles can be specified: start cycling at a specific time, terminate cycling at a specific time, execute with a designated periodicity while cycling, complete within a specified time during each execution.

In summary, RTC++ exhibits two faces: an object-oriented concurrent language and a real-time language. RTC++ adds the concept of active class to the usual concept of a C++ class. RTC++ supports only synchronous communication. In many ways, the specification of RTC++ appears superior to Concurrent C++ as proposed by Bell Labs researchers.

*Eiffel Concurrency.* Meyer proposes to introduce concurrency into Eiffel in a fashion that matches the style of the language very well and that requires (the more pessimistic might choose the word "allows") no explicit programmer control of concurrency, except specification of which Eiffel objects can be executed concurrently.<sup>52</sup> Meyer starts from two assumptions: 1) a concurrency mechanism should change sequential Eiffel as little as possible and 2) a concurrency mechanism must be

compatible with the Eiffel assertion mechanisms. These assumptions constrain Meyer's solution space.

Meyer introduces the key word **separate** as a qualifier that can be applied to Eiffel object declarations. An object declared as **separate**, we will call them active objects, may execute on its own (logical) processor. Meyer overloads the semantics of method preconditions such that, for active objects, evaluation of a precondition that is not satisfied means that the client (operation caller) cannot be served until the precondition is meant. Therefore, when an active object provides the service, an unsatisfied precondition does not cause an exception, rather the server is blocked until the precondition becomes true. Within the client, results from an active object may be assigned to another object declared active (and of a type that conforms in the Eiffel sense).

After a client invokes an operation, A, in an active object, the client can continue concurrently unless the client attempts to perform an operation the requires A to have been completed. If A must be completed for the client to continue, the client waits implicitly.

Meyer defines concurrency granularity to be at the level of operations. No operation may be interrupted, so each operation invocation is atomic. Meyer does envision library routines that can halt, under programmer control, execution of an operation. Of course, Meyer also foresees operations that can block interruption of an operation, in which case an Eiffel exception would be raised should an operation be interrupted.

In summary, the Eiffel concurrency scheme defined by Meyer is simple and elegant, relieving the programmer from most burdens usually associated with concurrent programming. (Of course, sometimes, the programmer can benefit from control levers, such as those defined in RTC++.) The scheme proposed by Meyer appears susceptible to deadlocks.<sup>63</sup> Meyer seems to have

swept most issues dealing with communications among active objects out of the language and into run-time libraries. We see no clear path from Meyer's concurrent Eiffel to a distributed Eiffel, except by making distribution conceptually transparent to the programmer.

#### *D. Other Models*

Many researchers are investigating models for concurrent and distributed objects. Some models concentrate on specific issues, and, so, are not easily classified. In this section, we survey five such models: 1) Trellis/Owl, 2) Emerald, 3) Hybrid, 4) Ellie, and 5) MELD. We chose these models because they each emphasized some significant issues relating to concurrency among and distribution of objects. Difference in emphasis leads to diversity in approach.

*Trellis/Owl.* The Trellis/Owl, hereafter Trellis, Language supports concurrency at the level of method invocations.<sup>53</sup> The intent of Trellis is to support moderate- to large-grained parallelism, where concurrency exists among logically separate tasks invoked independently by users. A given task can create new, subordinate tasks and can wait for subordinate tasks to terminate. Each task terminates with either a normal return or an exception. A task cannot be aborted, even by its parent task.

Trellis provides basic facilities for concurrent tasks to access shared objects, but the burden for managing that access falls on the programmer. Locks are implemented for mutual exclusion. Tasks attempting to reacquire locks are given priority over tasks that are attempting to acquire the lock for the first time. Reacquirers of a lock are ordered in accordance with the time they were awakened; this helps to prevent deadlocks. Wait queues are implemented for inter-task signaling

and for waiting on such signals. A task awaiting a signal may specify a time after which the task will be awakened if no signal is received.

Tasks can be interrupted, or swapped to disk, except when they are executing in a critical section, such as stack creation or deletion, calls to the run-time system, or initialization of global variables. All language-provided functions that operate on locks and queues are atomic.

In summary, Trellis defines an environment where multiple tasks can execute concurrently, but where little concurrency is supported within a task. The Trellis Language, and its associated run-time system, provides a sufficient set of constructs to control mutual exclusion and synchronization among concurrent tasks. The programmer must manage these mechanisms.

*Emerald.* Emerald is an object-oriented language and run-time system designed expressly to support the needs of distributed object systems.<sup>13</sup> Emerald introduces object location and mobility as explicit features. Objects can move between nodes at any time and can be invoked regardless of location. An Emerald object comprises an identity, a representation, and a set of operations. An object may be a process with an independent thread of control, or may be passive, executing only when invoked. Emerald objects have an explicit location attribute. Objects declared to be immutable can be replicated within the system of objects, allowing remote references to be resolved by object copying.

Emerald supports concurrency between and within objects. A process object has its own internal thread of control that is initiated upon object creation. At any time, multiple processes may be executing within a single object as a result of invocation of the object's methods by different processes. To permit mutual exclusion, objects can contain monitor sections,

i.e., sections where methods and variables are guarded by conditions.

Although all Emerald objects share the same conceptual model, three different implementations of that model exist. Standard types, such as integer, are implemented as a memory location, coupled with in-line operations. Objects contained within another object are implemented as compiler-allocated memory with the operations represented as procedure calls. Objects that can move around the distributed system and that can be remotely referenced by other objects are implemented in kernel-allocated data space. References to these objects are made via an object table.

Invoking objects remotely in the Emerald system can lead to some interesting effects. For example, parameters may be passed to a remote object by copying them. Unless the objects passed are immutable, the location of the object must be updated throughout the system. As a second example, an object may be moved to a remote node so that operations on objects at the remote node can be called locally. Of course, a programmer can explicitly request that specific objects be moved at any time. This scheme requires a global accounting of all objects and probably does not scale to large, distributed systems.

In summary, Emerald was designed specifically for distributed object systems. Objects can be invoked without regard to their location. The Emerald run-time system attempts to move objects around to most efficiently perform remote invocations. Programmers may also specifically move objects among nodes in the system. Emerald probably does not scale up to global, distributed systems.

*Hybrid.* Hybrid is an object-oriented programming language.<sup>55</sup> Hybrid objects can be either passive or active. Active objects communicate through remote procedure calls which always represent a transfer of control, or creation of a new

control thread, called an activity. Objects are organized into domains, i.e., processes. Multiple activities may execute within a domain.

Hybrid allows (forces) the programmer to establish the granularity of concurrency by defining each domain (true concurrency occurs only between domains). A domain serializes access to operations on objects within the domain. Activities represent the threads of control that can either: be executing in a domain or waiting on a queue.

In Hybrid, calls to operations may be delayed. Each operation can be associated with a delay queue that can be opened and closed by operations on the object. Each object will serve requests previously delayed before honoring newly arriving requests. (Operations without a delay queue are modeled as owning permanently open delay queues.) Delay queues serialize access to operations in an object; increased concurrency can be achieved through delegation.

Hybrid contains a delegate construct which can bracket an expression. Expressions within a delegate bracket can be executed asynchronously. When an object delegates a call, the calling object can proceed in parallel with the called object(s). Delegation can be used together with delay queues to implement constraints or invariants.

Hybrid permits even greater concurrency through the **coloop** and **coblock** constructs. Within these constructs new activities can be started and can then execute concurrently; the initiating object must wait until the newly created concurrent activities have completed before proceeding.

Mutual exclusion is provided in Hybrid by the **atomic** statement. A group of Hybrid statements bracketed within an **atomic** block may not be executed by more than one activity at a time.

In summary, Hybrid is a cross between an actor model and an object-oriented language, but without inheritance. Hybrid adds concurrency to object-oriented systems in a practical manner similar to ABCL/1, but Hybrid does not include features for priority interruption.

*Ellie.* Ellie is an object-oriented programming language that allows fine-grained parallelism.<sup>8</sup> Fine-grained in Ellie means that the smallest operations, such as multiplication and additions, can be implemented as parallel processes. Ellie also supports medium-grained (sets of operations) and coarse-grained (units comprising a large number of operations) parallelism. The grain size for parallel operations in a specific program is controlled by the compiler. The compiler will aggregate Ellie operations into larger objects when the target computer cannot efficiently support the fine-grained parallelism possible in Ellie programs.

Ellie supports an object model similar to that of Smalltalk, but Ellie objects are active processes. Ellie objects may either be operational (i.e., have side effects) or functional (i.e., immutable). Ellie provides object typing, genericity and polymorphism, and delegation and inheritance.

Parallelism and synchronization within Ellie are achieved by two forms of remote procedure call (RPC): bounded and unbounded. Bounded RPCs are normal synchronous object method invocation. Unbounded RPCs assign a result to a future object and allow the caller to continue. Later, when the caller must access the future object, the caller will block until the result is available. Ellie also includes a mechanism to synchronize access to local variables within an object after invoking a set of parallel operations. Synchronization delays are handled, transparently to the programmer, by the run-time system.

Ellie introduces a mechanism that enables objects to dynamically alter the operations exported at their interface.

Such a mechanism enables parallelism to be restricted or enabled under conditions determined during program execution. Restricting access to operations is the only means available for a programmer to control mutual exclusion, and, thus, a programmer must exercise great care when specifying synchronization among fine-grained, parallel processes.

In summary, Ellie extends object-oriented concepts to facilitate execution on massively parallel architectures. The Ellie compiler automatically adjusts the parallelism in a program to account for the limitations of a target computer. Alas, the programmer must bear the burden of ensuring correct synchronization among the fine-grained, parallel operations within a program.

*MELD.* MELD is an object-oriented programming language that provides encapsulated classes, multiple inheritance, and active objects.<sup>39,40</sup> MELD is designed to support concurrency at four levels: macro data flow, methods, objects, and transactions. MELD statements enclosed in a data flow block are ordered only when the output of one statement is among the inputs of another; otherwise, the statements are unordered and may execute in parallel. Multiple methods within an object may execute simultaneously, but the programmer may identify atomic blocks within an object. MELD objects can execute concurrently via synchronous and asynchronous message passing. MELD also allows transactions that cut across multiple methods and objects. Transactions appear to execute atomically and serially with respect to other transactions.

Within the address space of one MELD process, an arbitrary number of threads may execute concurrently. Each thread may operate across multiple processes (i.e., address spaces), with the local thread suspended until control returns from the remote process.

In summary, the design of MELD includes the full range of concurrent models encountered in the other approaches we surveyed. Most concurrent mechanisms within MELD, specifically data flows, object concurrency, and transactions, are controlled by the programming language. Concurrency among MELD object methods requires that the programmer specify operations within the object that must be executed atomically.

### III. Taxonomy of Issues in Concurrent and Distributed Objects

As concurrency is introduced into otherwise sequential programs, a new layer of issues is introduced. Further, because distribution implies concurrency, introducing distribution into a software system adds another layer of difficult issues. Coupling concurrency and distribution with object-orientedness can raise even more issues, not all of which are well understood. Below we present a taxonomy of issues regarding concurrent and distributed objects. Since issues related to concurrency are a subset of the issues related to distribution, we will begin with concurrency. Please be aware that, while many of the issues presented are interrelated, a taxonomy separates issues and classifies them for purposes of exposition.

#### *A. Concurrency*

Concurrency among threads of control that must coordinate operations raises some interesting, and for the most part well understood, issues. Such issues remain when an object orientation is superimposed on concurrency (or vice versa). An object orientation can, however, add an interesting twist to certain concurrency problems. Below, we identify, classify, and describe concurrency issues, and we introduce, where warranted,

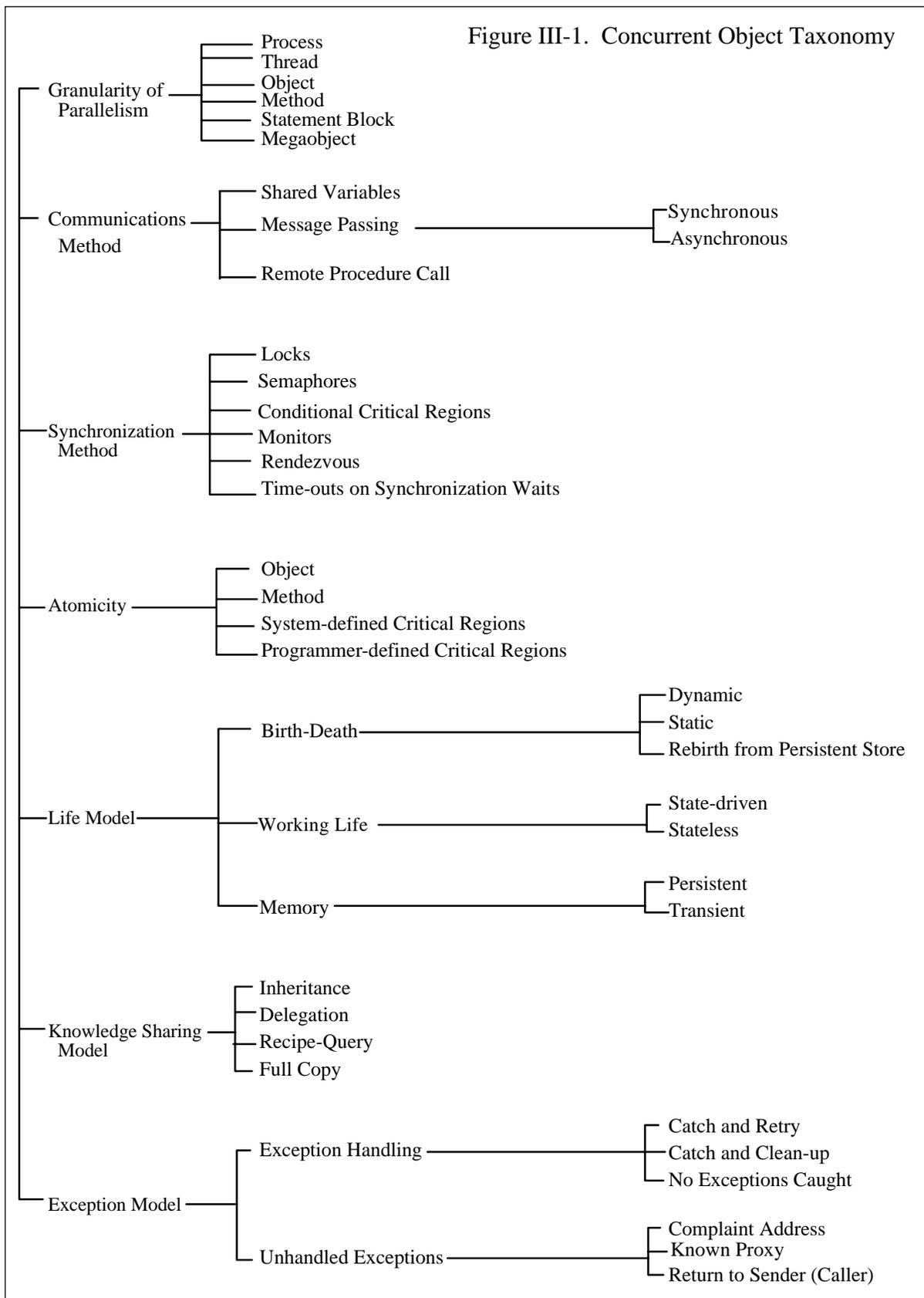
specific object-oriented concurrency concerns. A quick overview of the taxonomy is given in Figure III-1.

*Granularity of Parallelism.* The granularity of parallelism defines the limits of concurrency that can be achieved in a particular system. The traditional granularity for concurrency is the process or task. Each process has its own address space and can access memory, the processor, and other resources in competition with other processes. More recently, with the advent of several popular real-time operating systems such as VRTX and iRMX-86, granularity has moved within the process or task to the level of threads of control. Threads of control operate within the context of a process, sharing the process' address space, but being independently scheduled within the process. A thread of control can be viewed as a lightweight process. Switching between processes normally requires saving and loading memory management registers; switching between threads of control within a process requires only that the program counter and program registers be saved and loaded.

Within active object models, objects can have their own thread of control. In these models, objects might or might not share an address space with other objects. The mapping of objects and threads of control onto operating system resources varies with the specific active object model and the operating system environment. When objects define the granularity of parallelism, each active object can be independently scheduled, can compete for resources, and might require synchronization with other active objects. When an object is state-sensitive, access to the object must be serialized. When an object is immutable, access to the object may occur in parallel.

In a restricted set of active object models, the granularity of parallelism reduces to the level of methods. Some models serialize access to each specific method, while allowing parallel access among different objects. Some models

Figure III-1. Concurrent Object Taxonomy



serialize access to state-sensitive methods, but permit immutable methods to operate in parallel. Other models allow the programmer to open and close access to methods dynamically depending on the state of the object encapsulating the methods.

A few, recently designed object languages permit concurrency at the level of statement blocks. Most such languages enable the programmer to specify statement groups that can be executed in parallel. Some languages attempt to provide concurrency in a programmer-transparent fashion, enforcing statement sequence when the context requires and allowing parallel execution when sequential execution is not required. These languages rely on their compilers to perform the required analysis, but still seem to require care on the part of the programmer.

While much research into concurrent object systems aims to reduce the level of granularity (probably to accommodate massively parallel computing architectures), a few proposals (intended for application in large, heterogeneous computer networks) define concurrency at the megaobject level. Megaobjects encapsulate sizable services (often implemented as multiple processes, threads, or active objects), that are then distributed around a computer network, where they can provide remote services to a variety of clients. The megaobject is the level of visibility to clients in the network and, thus, defines the apparent level of concurrency for those clients.

*Communications Method.* Regardless of the granularity of parallelism in a system, the parallel units often need to exchange information to cooperate on a computation. Three general communications methods are possible: 1) shared variables, 2) message passing, and 3) remote procedure calls. We consider each of these in turn.

Shared variables require that concurrent units, wishing to communicate, be operating on a single processor, on a

tightly-coupled multiprocessor, or on a multiprocessor system with shared memory areas. Such arrangements enable processes to synchronize access to the same memory area where the processes can pass signals and data values to each other. Shared memory is a necessary prerequisite for a variety of synchronization schemes, such as semaphores, spin locks, monitors, and conditional critical regions. To accommodate distributed systems, shared memory must be replaced by a message passing mechanism.

Message passing comes in two general forms: synchronous, sometimes call tightly-coupled, and asynchronous, sometimes called loosely-coupled. Synchronous message passing requires that the sender of a message cannot continue after sending the message, but instead waits until the message can be accepted by the receiver. Two variations of synchronous message passing exist. One requires the sender to wait only until the receiver accepts the message. The other variation requires the sender to wait until the receiver replies to the message. (Another common variation allows the sender to continue unless the sending channel is full. This variation falls somewhere between asynchronous and synchronous message passing.) Synchronous message passing models usually require that communication be point-to-point. Synchronous message passing implies that message delivery is reliable, or else the sender might become deadlocked waiting for a reply.

Asynchronous message passing disconnects the activities of the sender from those of the receiver by supplying a queue to buffer messages until the receiver is ready to process them. The sender of an asynchronous message can transmit the message and continue processing regardless of the state of the receiver. When defining asynchronous message passing schemes three issues must be considered. First, the communications model must be established. Is each message sent to only one addressee

(unicast), or can messages be sent to groups (multicast) or to every addressee (broadcast)? Is each message an unrelated transmission, or is a reply expected for each message?

The second issue to consider when defining an asynchronous message passing scheme is the properties that the communications can exhibit. Will each message between a given sender-receiver pair be received in the sequence that it was sent? Can messages be lost, or are they guaranteed to be delivered eventually? If delivery is not guaranteed, can loss be detected? Corrected? Will every message be delivered as sent? If messages can be damaged, can the damage be detected? Corrected? Is every message guaranteed to be delivered once only? If duplicate messages might be delivered, can they be detected? Eliminated?

The third issue to consider when defining an asynchronous message passing scheme is the possibility of expedited, or express, messages. Can certain messages be designated to by-pass the normal communication channel between a sender and receiver, possibly jumping ahead of previously sent, messages? Can such expedited messages interrupt the receiver? And, of course for expedited messages, the issues of addressing model and communication properties must also be considered.

Another form of message passing, the remote procedure call (RPC), insulates the programmer from the fact that an invoked procedure is in inside another process' address space, and possibly in another computer on a network. In fact, the RPC is a restricted form of synchronous message passing. When a remote procedure is called, the caller yields control, just as though invoking a local procedure, but a library routine and kernel function must intervene to create a message, fill in the operation and parameters of the procedure call, and send the message, through an RPC client, to the correct RPC server. The RPC server receives the message, extracts the operation and parameters, and invokes the correct local procedure. When the

procedure is completed, the return call is routed through the operating system kernel to an RPC client, where a reply is created and returned to the correct RPC server. The RPC server will extract any return values and formulate a return statement, then the waiting process will be awakened and proceed from the point of suspension, as though the call had been to a local procedure.

*Synchronization Method.* For distributed systems, the only means of synchronization available is synchronous message passing or remote procedure calls; however, for general, concurrent processing systems a full range of synchronization methods may be built atop shared variables. Here we consider such shared-memory, synchronization mechanisms.

A simple, fine-grained synchronization method reserves a single variable, called a lock, that can be shared by all processes. Each process waits for the lock to be false, sets the lock to true, executes in the critical section, sets the lock to false, and continues processing outside the critical section. In the simplest incarnation, processes simply spin in a tight, busy-wait loop until they acquire the lock. Of course, when many processes are awaiting the same lock, a fair sharing algorithm is required to prevent the starvation of any one process.

Semaphores provide a more sophisticated basis for synchronization schemes than do locks. Locks induce busy-waiting loops that can be very inefficient, and locks require sometimes complex algorithms to ensure fair access. Semaphores combine wait queues with a variable to eliminate busy-wait polling and to provide the basis for a built-in fair access. The binary semaphore operates much as a lock, but without the busy-wait polling. A general semaphore allows the guard variable to take on positive integer values, ordering

processes on the wait queue to provide first-come, first-serve access.

Conditional critical regions (CCRs) extend the synchronization available with semaphores by providing simpler mechanisms that are easier to program. A named resource can be protected by embedding it inside a region block. Mutual exclusion can then be guaranteed by ensuring that execution of region blocks that name the same resource are never interleaved. In addition, conditional synchronization can be included by adding Boolean conditions onto the region statements. While CCRs are easier to program than semaphores, CCRs are also less efficient and, thus, are not widely used in practice. CCRs do, however, open a pathway to another synchronization method: monitors.

Monitors, passive guard modules, overcome the limitation that shared variables must be global to all processes in a system. Monitors can provide more structure than CCRs, yet can be implemented more efficiently than semaphores. Monitors encapsulate abstract resources and provide a specific set of operations visible to other processes that wish to access the resources. Mutual exclusion is provided by ensuring that execution of the operations within the same monitor do not overlap. In effect, the synchronization mechanisms are hidden inside the monitor's operations.

Another form of synchronization is the rendezvous. The rendezvous allows a task to await conditionally any of several events, or operation types. When one of the awaited events occurs, and any associated guard condition is satisfied, the processing associated with that event is invoked. If more than one of the awaited conditions is satisfied by an arriving event, only one is chosen nondeterministically. Sometimes, a scheduling clause is added to remove the nondeterminism.

All of the synchronization mechanisms we have discussed can lead to deadlock, if the condition or event awaited never occurs. To avoid such problems, synchronization schemes can be augmented with a time-out feature. In effect, a process waits for an event or condition, but specifies that it will only wait for period of time. When the period expires, the process is awakened even though the awaited condition did not occur.

*Atomicity.* To ensure consistency of shared information, a process may require that certain operations execute atomically. For example, on a computer, a typical machine instruction cannot be interrupted in the middle of execution, or, in an operating system, certain sequences of statements may execute with interrupts disabled. In concurrent object systems, four approaches exist to satisfy this requirement. One approach serializes access to an object, so that each operation invoked executes to completion before the next operation is accepted. A more liberal approach, supported in some systems, locks out only those operations that cannot safely execute in parallel with the current operation. For example, an object that is performing a computation to return a value might accept other access operations in parallel, but lock out access operations once an update operation begins. A third atomicity approach relies upon the compiler to analyze the code and determine which regions must be executed with exclusion and to generate appropriate instructions that enable the run-time system to enforce mutual exclusion where required. A fourth approach incorporates atomic blocks into a programming language. The programmer must then use these constructs to protect critical regions within the code.

*Life Model.* The life model of an object encompasses three aspects: birth-death, working life, and memory. A system of objects can be configured statically so that, when the system is loaded into computer memory, all the necessary objects exist and

are preconfigured. This approach might be useful for a real-time system with fixed capacities in memory and processing time. By preconfiguring a system of objects, there is no danger of exhausting memory and the system operates without the overhead of allocating memory and initializing objects. More common is a dynamic approach where all objects are created during run-time and destroyed when they are no longer needed. In fact, many object-oriented languages model program execution as creation of a root object; once the root object is completely created, the program is terminated. Some dynamic object systems enable all or subset of objects to be moved to persistent storage from which they can be recalled, state in tact, and executed in the future. The static approach is a restricted form of rebirth from persistent storage, except that each birth finds the object in the same initial state. Rebirth of a dynamic object system will find the objects in the same state as when they were moved to storage. Of course, there is no reason that a static object system cannot be interrupted, moved to persistent storage, reloaded, and resumed from the point of interruption.

While executing, an object can be stateless or state-driven. Stateless objects are analogous to mathematical functions. No concurrency protection is required for stateless objects. State-driven objects must be executed atomically and sequentially, unless some sophisticated scheme for concurrency is implemented by the language, run-time system, and/or programmer.

An object may possess persistent or transient memory. Persistent memory enables an object to exist across program executions, or to be restarted without loss of state after a system crash. Transient memory requires that each time an object is invoked initial conditions must be established.

*Knowledge-sharing Model.* Object-oriented systems enable new objects to be derived from existing objects by some form of knowledge-sharing. The usual form of knowledge-sharing in object-oriented systems is inheritance. When an object (child) inherits from another object (parent), the child obtains every attribute and feature of the parent. (A child can inherit from multiple parents, obtaining the union of the attributes and features of all the parents.) The child may then rename and/or redefine (subject to the rules applicable in the specific language) any of the features and attributes inherited.

Another approach to knowledge-sharing, delegation, occurs when one object knows about a proxy object.<sup>43</sup> At run-time, an object will delegate to its proxy operations that it does not understand. (This model assumes that the proxy is somehow more general, or has more knowledge.) If the proxy cannot understand the message, then it can delegate it to its own proxy, and so on. This message forwarding occurs at run-time, much as is the case for Smalltalk, leading to possible inefficiency when compared with inheritance (a compile-time mechanism).

The query-recipe scheme turns delegation on its head. Rather than delegating an unknown message, an object will ask its proxy for the recipe, or method, to process the message. This approach can lead to transmission of large messages. Some alternatives to sending the method from the proxy to the requester exist, but, in general, these alternatives lead to deadlock, which can be avoided only by sending many messages.<sup>17</sup>

A fourth approach to knowledge-sharing is to implement inheritance in distributed objects by simply copying all features and attributes from the parent to the child every time the child is created or compiled. (Note that under normal circumstances, where objects are in the same process, only references to methods need be copied during inheritance.) This will create large objects. Also, once the inheritance is

copied, any changes to the parent object will not be reflected automatically to the child.

*Exception Model.* Some object-oriented models allow certain exceptions to be caught. In general, when a model allows exceptions to be caught, the operation causing the exception can be retried after some action is taken to resolve the cause of the exception. Some models enable exceptions to be caught, but only so that the object can be returned to a known consistent state; in these systems, operations causing exceptions cannot be retried.

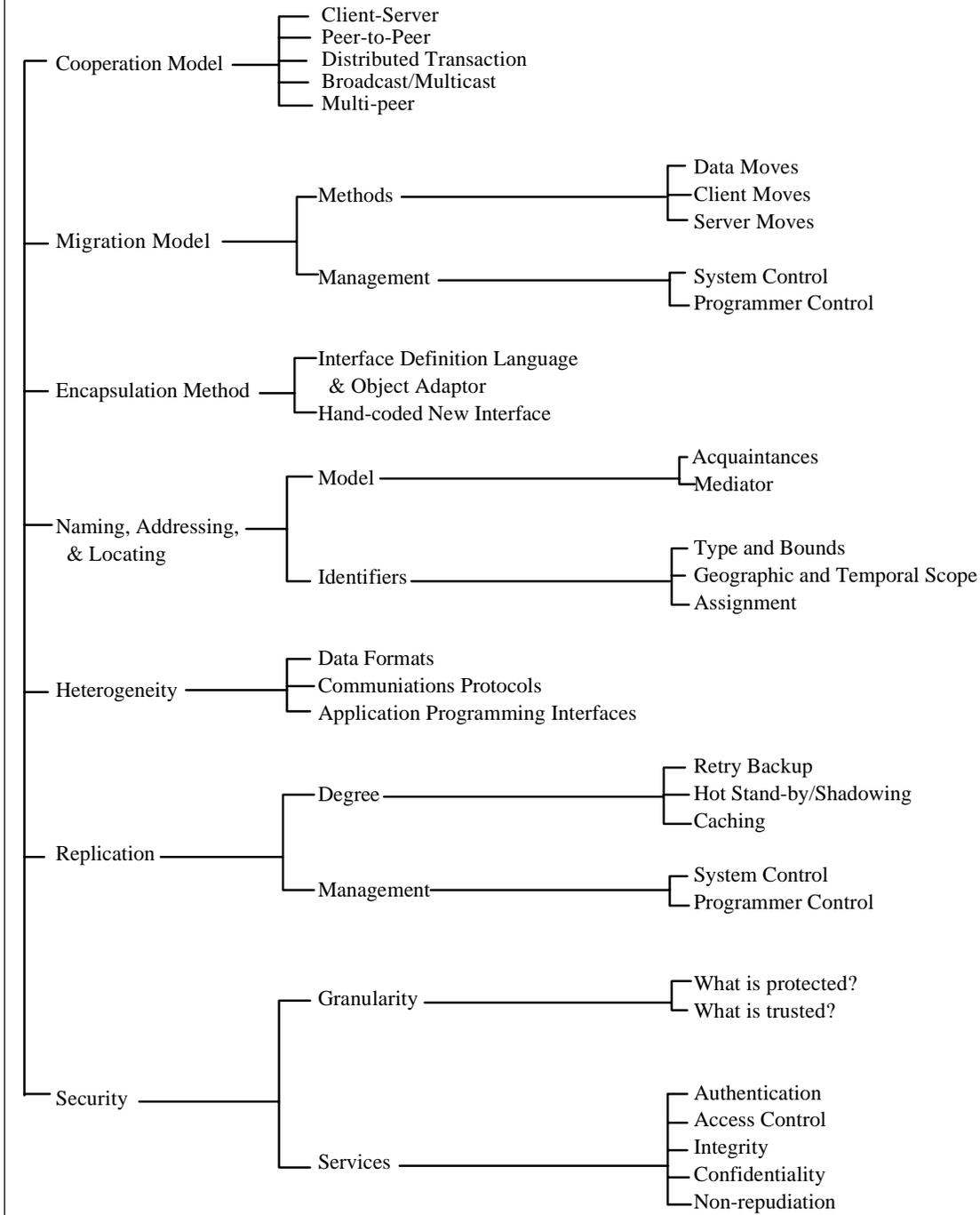
Whether an exception is unhandled because the programmer chooses not to catch it, or because no retries are allowed, or because the specific exception could not be caught, some notification of the unhandled exception is needed. Many object models simply notify the caller, or message sender, that the exception occurred. More sophisticated models allow the programmer to specify complaint addresses for each operation. When an unhandled exception occurs, notification is sent to the appropriate complaint address. A similar approach notifies a known proxy, specified for each object, when an unhandled exception occurs.

## *B. Distribution*

Distribution implies concurrency; thus, for a distributed object system, all of the issues we identified, classified, and discussed above apply. In addition, an equally large set of issues must be considered when a system of objects is distributed. These additional issues are illustrated in the taxonomy shown in Figure III-2, and are discussed below.

*Cooperation Model.* Within a distributed system, computation is accomplished through some form of cooperation between two or more active, independent processes. Various

Figure III-2 Distributed Object Taxonomy



models exist for cooperation among processes in distributed systems.

The client-server model extends function and subroutine calls across a computer network. A caller, needing to access a remote procedure, calls a local client which packages a message and sends the message to a remote server. The remote server transforms the message into a local procedure call. The return, or reply, flows back to the original caller. The client-server model is easy for programmers to understand and work with, but other models can provide more flexibility in a distributed environment.

The peer-to-peer model treats communication between two distributed processes as an interactive dialog. Generally, two processes establish a connection with each other and then exchange data in both directions simultaneously. Peer-to-peer communication, similar to human conversation across a telephone connection, is efficient when large quantities of data must be exchanged between two processes, for example, during a file transfer operation, or when two processes must interact quickly over a prolonged period, for example, during a process control application. The peer-to-peer model, while more flexible than the client-server approach, limits communication to two parties.

The distributed transaction model enables a process to interact simultaneously with many other processes in a distributed system. A master process, responsible for processing a transaction, can interact with many other processes, each being delegated a portion, or subtransaction, of the job. The master process must ensure that either all subtransactions are completed or that none of the subtransactions is completed. In effect, the master process ensures that the transaction is atomic, consistent, indivisible, and recoverable. As a general approach, the distributed transaction model is recursive, so every subtransaction may

spawn additional subtransactions, forming a distributed transaction tree with the initiating process at the root. Distributed transactions are cumbersome, complex, and inefficient, but they provide a huge increase in functionality when compared with either the client-server or peer-to-peer model.

Two other cooperation models are sometimes used, although in restricted applications. The broadcast model, where a processes sends one message that is copied to every other process in the distributed system, is sometimes used to distribute global events, or to distribute information to everyone, in the absence of an explicit destination address, expecting the intended addressee to process the message and others to ignore it. Probably the most common use of broadcast, limited to a local area network segment, is address resolution. Address resolution is required when a message, containing a destination name, arrives from outside a local network. The receiving node, if the destination name is unknown, will broadcast a small message on the local area network asking: "Does anyone recognize this destination name?" If a specific node recognizes the name, the node replies: "Send any messages for the destination name to this address." Because the broadcast model does not require a reply, broadcast messages need not arrive at every destination. In the context of wide-area networks, broadcast operations are expensive, so practical use of the broadcast model is limited to local networks. A similar, but more efficient, multicast model can be applied outside of local networks.

The multicast model enables groups of processes to be formed and then labeled with a group, or multicast, address. Whenever a message is sent to the group address, everyone in the group will be given a copy of the message. (When a group includes every address in the system, then the multicast model

is the broadcast model.) The multicast model is sometimes used in electronic mail applications (to embody mailing lists) or in command and control applications. As with broadcast, multicast messages are not guaranteed to be delivered. A more reliable model, multi-peer, is the subject of research.

Multi-peer communication enables a process to establish a single communications channel with multiple destinations. In one multi-peer model, each message sent by a designated channel master elicits an acknowledgment from the multiple slaves. This model has been implemented in several experiments involving one-way, reliable, multicast transmissions over satellite links. The general utility of the model has yet to be proven.

*Migration Model.* In object-oriented distributed systems, knowledge must be shared between objects that may be remote from each other. For example, an object may need to invoke a method in a remote object, or access data from a remote object. Rather than placing objects in a fixed location and then relying solely on remote procedure calls for information sharing, some recent research seeks to address these issues by allowing objects to move around in a distributed system. Several movement strategies, or migration models, are being investigate.

One migration model replicates object class code throughout the network and then moves specific instantiation data between nodes when remote access is required. This approach eliminates movement of large code segments during run-time (the code must be distributed after each compilation) and can ease the problem of transformations required to move information in a heterogeneous network.

A second migration model moves a copy of the client object to the node where the appropriate server is operating. This approach is usually limited to a homogenous network. A related approach moves the server object to the node where the client resides.

No matter what migration model is advocated, the actual movement of objects can be controlled by the system or managed by the programmer. Neither of these control approaches has proven superior to the other, because moving objects around in a system of distributed nodes, no matter how the movement is managed, remains a research problem with many complex aspects. What will be the performance effects of moving objects around the network? What happens when each object in a series of calls is on a different node? When can objects be replicated? How will the global state of all object locations be reflected? How will heterogeneity of computer architectures, object languages, and run-time systems be accommodated? We find object migration to be an active research area.

*Encapsulation Method.* When an existing service, available in a network, must be incorporated into a distributed object model, a method must be selected to encapsulate the existing service inside of an object. At present, two methods exist: 1) a new interface to the existing service can be hand-coded in some object-oriented language or 2) a new interface to the existing service can be defined using an interface definition language, a supporting compiler, and an object adapter. The first approach is well understood, but limited because a new interface must be hand-coded for each new object-oriented language that wishes to access the existing services. The second approach, provided the interface definition language is a widely supported standard, allows the mapping between an existing service and an object-oriented interface to be carried out once. Then, clients wishing to access an existing service can encode their interface using the same interface definition language. To allow the interfaces to be supported in multiple object-oriented languages, object adapter library routines must be written for each object-oriented language. Certainly, encoding the object adapter routines once for each

object-oriented language is more efficient than hand-coding an interface in every object-oriented language for each existing service that must be encapsulated.

*Naming, Addressing, and Locating.* To access objects in a distributed system, each object must have an identifier that is unique within the scope of the system. Further, when an object's identifier is known, a means must exist to locate a specific object corresponding to the known identifier. Often, a distributed system is limited by the form and semantics of the identifiers available in the system.

Objects can be identified by a name, an address, or a type reference. A name identifies an object independent of its location. An address identifies an object implicitly by specifying where the object is located. Normally, when an object name is known, the name must be turned into an address. From an address, the distributed system should be able to locate the named object. In most object systems, a type reference identifies an object within a certain class. Usually, a type reference can be resolved into the address of a specific object of the named type. In practical terms, an object is requested by name or type reference when the object is created, and an address is returned from the creation call. From that point on, the object is referenced by the address. (Of course, in systems where objects move, the address may be logical rather than physical.)

Whether names, addresses, or type references, identifiers exist within a geographic, or topological, scope and a temporal scope. Within the bounds of the system geography, an identifier must be unique; names and addresses must map to objects on a one-to-one basis and type references must map to classes on a one-to-one basis. In general, names and type references are long-lived, while addresses live only from the time an object is instantiated until the object is moved or destroyed.

Within a distributed system, a means is needed to obtain and assign identifiers. Whenever an object or object class is declared, a unique name or type reference, respectively, must be obtained and assigned to the object. Whenever an object is instantiated, a unique address must be obtained and assigned to the object instance.

As if obtaining and assigning names and addresses in a distributed system were not difficult enough, once the identifiers are assigned, some mechanism must enable the knowledge of specific identifiers to be shared. One such mechanism encodes a set of known acquaintances into the declaration of each object. This initial set of acquaintances permits the object to exchange messages with other objects after instantiation. While exchanging messages, an object may make new acquaintances, thus widening the scope of its community. When acquaintance lists are used, the identifiers must be addresses. A less restrictive mechanism relies on a mediator.

A mediator can be queried with an object name or a type reference. The mediator will lookup the appropriate object that matches the query and then, providing the name or reference is valid, return the associated address to the requester. The mediator might be a name resolver, a directory server, or an object request broker.

*Heterogeneity.* When a distributed system consists of identical computers, running the same operating system, an entire class of issues can be ignored; however, in the more general case of heterogeneous computer systems, incompatible data formats must be resolved, incompatible communications protocols must be eliminated, and applications programming interfaces (APIs) must be selected to enable application programs to be moved between computers. These issues can best be resolved by establishing standards for data formats, communications protocols, and APIs. Fortunately, progress is

being made toward standards in these areas. Unfortunately, most researchers who are investigating distributed objects ignore the issues surrounding heterogeneity, assuming instead that all computers on their networks are homogeneous. Researchers within the computer industry who are investigating distributed object environments assume that heterogeneity is a primary concern that must be addressed.

*Replication.* Within distributed systems, replication of information is used to decrease network congestion, to increase responsiveness, and to increase fault tolerance. By mirroring a copy of system memory on a disk, a node can recover and restart from the point of interruption after a crash. In more critical applications, one or more nodes can be assigned to shadow another node so that, should the shadowed node fail, a shadowing node can pick up the processing responsibilities immediately and correctly. This mode of operation is sometimes called hot-standby.

A simple strategy to improve responsiveness allows each node to cache information, obtained remotely, so that subsequent calls can be processed from a local data store. Caching approaches raise a number of issues, such as the granularity of information that should be passed with each remote request, the method of determining that cached information may be invalid or stale, and the method for updating cached information held throughout the network.

No matter which specific replication scheme, mirroring, shadowing, or caching, is used, the process must be managed. Some systems enable the programmer to manage replication, while others build replication management into the run-time system.

*Security.* System security, an issue for any computer operating system, takes on increased importance in a distributed environment because each computer in a network is open to attempted access by unknown users and because messages sent

between computers may be altered or copied by unauthorized users. To protect distributed computing systems, two issues must be decided: 1) the granularity of protection and trust and 2) the specific security services provided.

In a system of distributed objects, the issue of protection granularity is interesting. Should access to an object be protected or should access be restricted on a method-by-method basis? Should messages between objects be protected or should protection be restricted to specified parameters in the messages? Should the object nature of the distributed system be ignored, leaving security granularity at the level provided for any system on a network?

An issue that complements the granularity of protection is granularity of trust. Given some decisions about what is to be protected, what are the entities that should be authenticated, and then trusted according to that authentication? In distributed systems, processes can be authenticated on behalf of users, network nodes can be authenticated on behalf of processes running on the node, and specific communications between nodes or between processes might be authenticated individually. What about an object-oriented distributed system? Should each object be authenticated, or are the usual levels of authentication in a distributed system sufficient?

Complementing the issues of granularity is the question of what services to offer. Should authentication services be based on public keys, or should a third-party authentication service issue, in real time, private keys? Should access controls be provided at multiple levels, as opposed to just access okay or access denied? For example, for a specified, target object, should user objects authenticated as system auditors be given access to special methods? If messages between objects are protected, should protection include both integrity and confidentiality? Should these protections be applied to entire

messages, to specified fields, or to programmer designated portions of messages? Should these protections be applied to every message, to programmer-designated exchanges, or to system selected transmissions? Should audit services be provided so that a series of object creations and method invocations can be reconstructed later?

#### IV. Applying Objects To Communications Architectures

We draw several implications from the foregoing survey of approaches to, and analysis of issues surrounding, concurrent and distributed object systems. First, no existing model or approach to concurrent, distributed object systems is sufficiently advanced for operational deployment. Further, the breadth and complexity of the issues that must be solved before a general model of concurrent, distributed object systems is accepted leads us to conclude that operational deployment of such systems will occur later, rather than sooner. For these reasons, we expect that conventional communications architectures among loosely-coupled, heterogeneous computer systems will play a growing operational role, as the number of computers and networks deployed continues to increase. This reasoning leads us to consider how object-oriented techniques might be employed to reduce the complexity, to improve the reusability and extensibility, and to increase the performance of convention communications architectures.

In the following sections we demonstrate how the concept of abstract data types (ADTs) can be used to specify the services provided by a layer within the Open Systems Interconnection (OSI) Reference Model. We use as our example the OSI transport layer, the fourth of seven layers described in the OSI Reference Model. We take as our starting point the OSI

Connection-oriented Transport Service Definition<sup>33</sup>, a 31-page international standard (IS 8072) that describes the OSI connection-oriented transport service through text, tables, state diagrams, and event sequence diagrams. We extracted the semantics of IS 8072 and represented them, in only 11 pages (see Appendix A), as an ADT using the Eiffel notation.

While the Eiffel ADT successfully captured the semantics of IS 8072, we found the result to be impractical for use as an application programming interface (API). The ADT captures some details that would normally be hidden from an application programmer. The ADT also is restricted to interactions between a single transport service user and a transport service provider. This client-server relationship is somewhat unrealistic because the transport service involves peer-to-peer interactions between two users through the transport service. Figure IV-1 illustrates this issue.

Each instance of the OSI connection-oriented transport service is provided over a full-duplex connection that is established between two users. In Figure IV-1, the full-duplex channel between two transport service users is represented by two simplex channels, one from User A to User B and the other from User B to User A. IS 8072 requires that such a connection be established by only one of the users (the connection

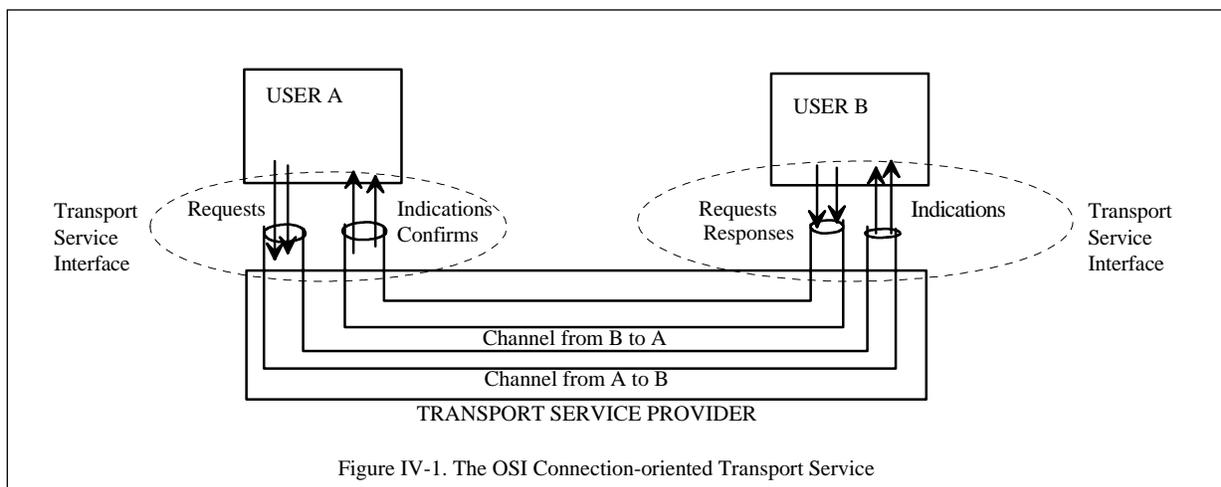


Figure IV-1. The OSI Connection-oriented Transport Service

initiator). In Figure IV-1, User A would send a connect request (CR) to User B which would arrive at User B as a connect indication. User B, if accepting the connection, then issues a connect response which arrives at User A as a connect confirm. Once the connection is established, the Users operate independently and symmetrically with each issuing various data requests which arrive at the other user as data indications. The interactions can continue until either user decides to end the connection by issuing a disconnect request, which arrives at the other user as a disconnect indication. The Transport Service Provider relays the requests and responses between the users and ensures that various service guarantees are achieved.

An ADT for the OSI transport service specifies the interactions between a user, or client, and the transport service provider, or server. As can be seen in Figure IV-1, two transport service interfaces exist, one between User A and the provider and the other between User B and the provider. A user may access the transport provider in either of the two roles, but not in both simultaneously (unless the user operates in loopback mode, in which case the user is still acting as two users and would have to be specially constructed).

An application programming interface (API) might be concerned not only with interactions between the provider and the user, but also between the local and remote user. In fact, the API might be interested solely in the interaction between the two users. The ADT given in Appendix A does not capture the interaction between users. For this reason, we defined a second Eiffel description of the transport service (see Appendix B) that is intended to provide an API.

Both the ADT and API were written in Eiffel (version 2.3) and compiled on a UNIX system. We then moved the compiled code to a Microsoft Windows system and used word-processing software to improve the appearance of the code. In both Appendix A and

B, we show the Eiffel keywords in **bold**. Classes that we have defined are shown in **CAPITAL, BOLD, ITALICS**. (Built-in Eiffel classes are shown in **BOLD CAPITAL** letters.) We assigned line numbers to each listing so that the reader can relate the following discussions to the appropriate statements in the Eiffel code.

We first discuss the ADT for the OSI transport service interface. We then describe the API for OSI transport users.

#### *A. An Abstract Data Type for the OSI Transport Service Interface*

Our abstract data type (ADT) for the OSI transport service interface is shown in Appendix A as the Eiffel deferred class **TS\_INTERFACE** (lines 1 to 285). Each exported feature, except `expedited_allowed`, corresponds to an abstract transport service primitive defined in IS 8072. The `expedited_allowed` (line 22) feature indicates whether or not expedited data can be sent across the transport connection. (Each transport connection can send regular data, but the possibility also exists to send expedited data which can leapfrog the normal flow control associated with regular data. Expedited data facilitates sending of interrupts across a transport connection.)

IS 8072 defines a small (four states and eight transitions) finite state machine (FSM) that controls the types of service primitives that may be issued by the transport service user and provider at any point during the life of the connection. This FSM is represented by the hidden feature `TS_Interface` (line 14) of the class **TC\_STATE** (lines 289 to 311). The transport service interface can be in one of four states (idle, outgoing connection pending, incoming connection pending, or data transfer ready). The state of the interface is used in preconditions to the service primitives to prevent the primitives from being issued at the wrong time.

The hidden features now (line 16) and `time_sent` (line 18) are used to provide time-stamps for certain operations and to access the current time. (One, perhaps unrealistic, assumption made by us is that a global clock exists and can be accessed to provide synchronized time.) `TS_User_Invoked` (line 20) provides a value that can be inserted into the reason field of a disconnect request when the transport service user initiates a disconnect. This allows user-initiated disconnects to be distinguished from provider initiated disconnects.

IS 8072 describes the transport interface as consisting of a full-duplex channel (usually represented as two simplex channels running in opposite directions) between two transport service users. IS 8072 also specifies that certain requests issued by a transport service user can overtake and even cause deletion of other requests while they are still in the channel. The rules describing how this can occur also specify that these operations are optional and under control of the service provider. To allow our ADT to represent these rules, we modeled the channel between the transport service user and provider as six simplex channels (three for inbound objects and three for outbound objects), as shown in lines 24, 26, and 28. The **CHANNELS** (lines 330 to 345) `cd_in` and `cd_out` contain **CTL\_BLOCKS** (lines 347 to 373) which include connect requests (CRs), connect confirms (CCs), and disconnect requests (DRs). In effect, these are connection management messages. The DRs can potentially destroy normal and expedited data that is en route to transport service users, but not yet delivered.

The **CHANNELS** `dt_in` and `dt_out` contain normal data, called transport service data units (**TSDUs**, lines 377 to 394) sent by and arriving for, respectively, the transport service user. The remaining pair of **CHANNELS**, `ed_in` and `ed_out`, contain expedited data (**E\_TSDUs**, lines 397 to 412) leaving from and arriving for the transport service user.

All messages crossing the transport service interface may, and some must, contain data which we represent as a **DATA** class (lines 415 to 421) consisting of a count and value (**ARRAY[CHAR]**). Each type of message we represent (**CTL\_BLOCK**, **TSDU**, **E\_TSDU**) contains parameters in addition to **DATA** -- the specific parameters, represented as features in the appropriate class, correspond to parameters defined in IS 8072 or to parameters needed to guarantee the service defined in IS 8072. We will cover each parameter, as necessary, in the subsequent discussion.

The transport service interface is initiated when a transport service user, hereafter **USER**, invokes the **T\_CONNECT\_request** feature (lines 30 to 57) to attempt to establish a connection, via the transport service provider, hereafter **PROVIDER**, to a remote user. We will describe this feature definition in some detail so that the reader may understand the major points of the remaining features on his own.

The first parameter in the **T\_CONNECT\_request**, **Invoker**, (line 30) is an invention of our own. **Invoker** is of class **IDENTITY** (lines 313 to 327) which simply distinguishes the service provider (**isPROVIDER**) from the service user (**isUSER**). This is necessary because IS 8072 indicates that certain of the service primitives (represented by us as Eiffel features in the class **TS\_INTERFACE**) may only be issued by the service user and others may only be issued by the service provider. In effect, our ADT is an entity that is shared by a transport service user and a service provider -- we have represented **TS\_INTERFACE** as a server to multiple clients, each of which is allowed to use certain features in a controlled manner.

The remaining parameters in the **T\_CONNECT\_request** are extracted directly from IS 8072. We have represented the Called

and Calling fields as an **ADDRESS** class (lines 455 to 459) and the Quality\_of\_Service fields as a **QOS** class (lines 430 to 434).

Examining the preconditions for the T\_CONNECT\_request (lines 37 to 42) we find that the invoker of this feature must be the USER, that the interface must presently be idle, that the called and calling addresses and quality of service must be provided, and that the user data is optional, but if present must be between 1 and 32 bytes in length. With the exception of the invoker identity (previously explained) all of the preconditions represent specifications from IS 8072.

The post-conditions for the T\_CONNECT\_request (lines 46 to 55) specify what the feature ensures, given that the preconditions were satisfied. The main post-conditions specified fall into three categories: 1) a **CTL\_BLOCK** is placed in cd\_out, 2) the expedited option requested by the USER is recorded in the expedited\_allowed feature, and 3) the interface state is changed to outgoing connection pending. Regarding the object placed into cd\_out: the count of objects in the channel is increased by one, the type of object is a CR, and the USER provided parameters are mapped into the CR object.

The reader should be able to follow the other features related to connection establishment. The T\_CONNECT\_indication (lines 59 to 82) is issued by the PROVIDER to signal an arriving CR. The preconditions require that the first object in cd\_in is the CR and that all parameters were mapped from that CR to the feature call. The post-conditions indicate that the CR was removed from cd\_in, that the incoming expedited option is saved, and that the interface state is changed to incoming connection pending.

After receiving a T\_CONNECT\_indication and deciding to accept that connection, the USER issues a T\_CONNECT\_response (lines 87 to 116). The interpretation of the pre- and

post-conditions is similar to that for a T\_CONNECT\_request, but a CC object is placed in cd\_out.

When the PROVIDER issues a T\_CONNECT\_confirm (lines 118 to 142) to a USER, the interpretation of the pre- and post-conditions follow along the lines of that given for the T\_CONNECTION\_indication. The result of the T\_CONNECT\_confirm is that the USER is in the data transfer ready state and that the expedited option for the connection has been finally established (this is defined in IS 8072 as a simple negotiation).

Once in the data transfer ready state, a USER may invoke repeatedly the T\_DATA\_request feature (lines 144 to 162) and the PROVIDER may invoke repeatedly the T\_DATA\_indication feature (lines 164 to 185). Each T\_DATA\_request requires that some associated **DATA** exist. The post-conditions ensure that the outgoing **TSDU** is stamped with the time, numbered in the sequence sent and placed in dt\_out. The time-stamp and sequence numbers are checked in the preconditions of T\_DATA\_indications to verify that the PROVIDER yielded the service specified in IS 8072 (i.e., the data will be delivered, after some finite delay, in the order sent, without damage, without gaps, and without duplication). The post-conditions for the T\_DATA\_indication ensure that the arriving data is removed from dt\_in.

If expedited\_allowed is true, then the USER may issue T\_EXPEDITED\_DATA\_requests (lines 187 to 208) and the PROVIDER may issue T\_EXPEDITED\_DATA\_indications (lines 210 to 234). The main outline for these features is taken from the corresponding T\_DATA\_request and T\_DATA\_indication features, but two wrinkles are added (because of requirements included in IS 8072). First, expedited data must be between 1 and 16 bytes in length. Second, any expedited data received must be received before any normal data sent after the expedited data. (Frankly, we find this definition of expedited to be less than satisfactory -- in effect IS 8072 guarantees that expedited data won't get treated

any worse than normal data, but *encourages* the PROVIDER to push the expedited data ahead of any normal data that is not yet delivered.)

The final features, T\_DISCONNECT\_request (lines 236 to 259) and T\_DISCONNECT\_indication (lines 261 to 283) deal with connection termination. A DISCONNECT\_request, which may optionally include data, may be issued by the USER when the interface is in any state except idle. If the request includes data, the data must be between 1 and 64 bytes in length. At the end of the T\_DISCONNECT\_request, the post-conditions ensure that: all incoming channels (cd\_in, dt\_in, and ed\_in) are empty, that the interface is idle, that any objects already in outgoing channels (cd\_out, dt\_out, and ed\_out) might be deleted (but they need not be because IS 8072 leaves this to the PROVIDER's discretion), and that the last object in cd\_out is a DR with a reason\_code of TS\_User\_Invoked.

The T\_DISCONNECT\_indication can be invoked by the PROVIDER at any time when the interface is not in the idle state. The T\_DISCONNECT\_indication ensures that the interface is idle and that all incoming and outgoing channels are empty.

These twelve features comprise the **TS\_Interface** ADT, an ADT that captures the OSI Transport Service Definition as embodied in IS 8072. While this ADT works well as a specification of the transport service, the odd arrangement of two clients (the USER and PROVIDER) communicating through one server (the ADT) provides an unconventional application programming interface (API). For this reason, we specified the OSI Transport Service Definition, again using Eiffel, in a different style that is more conventional for programmers. In addition, we tried to capture the relationship between actions taken by the two corresponding users across the transport service. At the same time, we dropped some of the details, such as quality of service, expedited data negotiation, and user data in the

connection management messages, that, although included in IS 8072, are not normally implemented. Our attempt at an API for the OSI transport service, included as Appendix B, is described below.

## B. *An Application Programming Interface for the OSI Transport Service*

Our application programming interface (API) for the OSI transport service is shown in Appendix B as two main classes: **TS** (lines 1 to 216) encapsulates the transport service provider and **TS\_USER** (lines 223 to 315) encapsulates the transport service user. We begin our discussion with **TS** because this class provides the programmer's view of the transport service.

**TS** exports seven features, each corresponding with a service offered by the provider to a user, or client. The parameters provided for each feature are minimal so as to ease the programmer's job. The listen feature establishes that a user is willing to wait for possible incoming connections. The connect feature initiates an active connection attempt by the transport service provider on behalf of the user. The disconnect feature terminates the connection. The send and receive features transmit data and accept any incoming data, respectively. The send\_expedited and receive\_expedited are analogous to send and receive, but operate on expedited data.

A user wishing to accept an incoming transport connection invokes the listen operation (lines 17 to 33) with an input parameter of class **TS\_USER** (lines 223 to 315). The listen operation requires that the input user is a valid user and that the user is disconnected, and then ensures that the user is listening or else is connected to another user who issued a connect that arrived while the listening user was being registered.

To actively establish a connection, a user invokes the connect operation (lines 35 to 57), specifying the local and remote users; both must be valid. The local user must be disconnected and the remote user must either be listening, disconnected, or trying. (Our API allows for two users attempting active connections to be resolved into a single connection. IS 8072 forces such a situation to resolve into two connections; thus, our API is more flexible than our ADT on this point.) The connect operation ensures that either the local and remote users are connected or that the local user is disconnected.

The disconnect operation (lines 59 to 79) allows the user to terminate an existing connection. The user must be valid and connected. The remote user must either be connected or be disconnecting. (Here again, our API is more flexible than our ADT. IS 8072 requires that data be neither sent nor received by a user after a disconnect request is issued, but our API, while preventing a user from sending data after issuing a disconnect, allows a user to continue receiving data after issuing a disconnect. This choice was made with the view that the same API might be used over multiple transport services each of which operates with slightly different rules.)

The send operation (lines 81 to 107) requires a user and data (of class *TSDU*, lines 318 to 342). The user must be valid and connected and the data must exist. The send operation will then ensure that the order in which the data is sent will match the order in which it is received, that the data is received after some finite delay, and that the content received matches the content sent; or else the user is disconnected. These assurances match the service guarantees of IS 8072 and can be expressed in compilable Eiffel; however, these assurances cannot be checked in reality because the send operation is not required to wait until the data is received but can continue, including

sending additional data. The transport service forms a pipeline of data that will eventually guarantee an outcome, but will not provide an outcome at the time of the call. Thus, the ensure clauses for the `send` and `send_expedited` operations should probably be expressed as comments.

The `receive` operation (lines 109 to 133) requires a user and returns a ***TSDU***. The user must be valid, and connected or disconnecting (this situation was described earlier). The operation ensures that the resulting ***TSDU*** is void if no data is waiting to be received. If data is waiting, the returned ***TSDU*** is guaranteed to have been sent earlier, to be received in the order sent, and to have the same content that was sent. Here, with some small changes to the transport protocol, the first two assurances can actually be checked, but the latter assurance cannot practically be evaluated. (Again, part of the ensure portion of the `receive` and `receive_expedited` might best be handled as Eiffel comments.) If the expectations of the transport service would be violated by a `receive` operation, then the user will be disconnected. The `receive` operation is non-blocking, that is, whether data is ready to be received or not, control will be returned immediately to the caller.

The operation of the `send_expedited` (lines 136 to 166) and `receive_expedited` (lines 168 to 195) features mirror those of the `send` and `receive` features, respectively. One additional wrinkle deals with the ordering of the expedited data relative to normal data. Any expedited data sent is guaranteed to be received prior to any normal data that is sent after the expedited. Note also that the expedited data sent and received is represented by a class ***E\_TSDU*** (lines 344 to 360).

The second major class, ***TS\_USER*** (lines 223 to 315), comprising the API represents the transport service user. In our description we included a number of details, as will be explained, and omitted others, such as addresses, quality of

service, and expedited flags. The omitted details were dealt with under the ADT and, thus, the reader should be able to supply them. Rather than passing the detailed parameters individually, as specified in IS 8072, we pass a reference to the user object that encapsulates the detail parameters associated with the user.

Each user may be in one of five states; therefore, we export five features querying the state of the **TS\_USER**. We also provide features indicating whether data has been sent and received, as well as the last sequence number of each **TSDU** sent and received. The wakeup operation allows the transport service provider to alert the user that some data or event has arrived that may be of interest to the user. The wakeup operation, then, is a suffered operation that must be implemented by the transport user in order to use the transport service. The only parameter of the wakeup operation is a reference to a local image of the remote user with whom the local user is communicating.

Two hidden operations are provided to allow the user to update the sequence numbers of the **TSDUs** sent and received. The Zero feature permits the invariant to compile. The invariant for **TS\_USER** is straightforward: sequence numbers must be zero or greater, if a **TSDU** has been sent or received, then the corresponding sequence number must be positive, and the **TS\_USER** must be in one of its valid state.

### *C. Evaluation*

The API we described is simpler, yet more flexible, than our ADT for the equivalent transport service, but also less precise. Since most of the post-conditions cannot really be evaluated at run-time, the API would probably require commenting out the ensure clauses for several of the operations. We

imagine that additional thought might lead to a better expression of the API, or, perhaps, the ADT can be adapted to become an API.

As a vehicle for specifying the OSI Transport Service, ADTs, particularly as represented by Eiffel, worked well. Thirty-one pages of text, pictures, tables, and state diagrams were condensed to eleven pages of Eiffel. Some of the behavior that is described as optional in IS 8072 could only be hinted at indirectly in the ADT. For example, the ability of some objects to overtake other objects at the discretion of the service provider could not be readily expressed in the ADT. In situations where ambiguity must be introduced into a specification, natural language is superior to the mathematical rigor of ADTs. Even so, the ADT we created for the transport service defines, almost completely, a fairly complex object.

As for using Eiffel to define an API to a distributed, peer-to-peer service, our success is less clear. We find that the preconditions are useful to constrain calls on the various operations; however, we believe that the post-conditions, for the most part, cannot be realized in an actual run-time implementation and, therefore, must be included in the API code merely as comments to describe what the programmer can expect. Of course, this may not be viewed as a limitation by Eiffel adherents because the recommended compile-time options under normal execution do not include the evaluation of post-conditions.

We found our experiment with ADTs and Eiffel to be stimulating. More thought will be required to determine if object-oriented techniques can be used to improve the specification of communications architectures, protocols, and services. Our early test showed some promise, as well as the need for additional learning on our part.

## V. Conclusions

We have considered three trends in information technology:

- 1) an increasing number of cheap, powerful computing devices,
- 2) a growing web tying these computing devices together, either into loosely-coupled local and wide area networks or into closely-coupled, massively parallel computational engines, and
- 3) a movement toward object-oriented software systems.

Combined, these trends seem to foretell an era of distributed, and thus concurrent, systems of objects, objects spread throughout a network of computers: communicating, moving, locating one another, fending off unauthorized accesses, cooperating to satisfy application requirements, synchronizing concurrent accesses, and handling exceptions that occur remotely. Our survey of approaches to concurrent and distributed object systems convinces us that this impending era has not yet arrived. No distributed object system is deployed today.

Further, our assessment of the issues facing any deployable, distributed object system convinces us that an era of distributed object systems will come later, rather than sooner. No consensus exists as to the cooperation model to be used among distributed object systems. Methods for naming, addressing, locating, and moving objects in a global network have yet to be established. Most present research on distributed object systems ignores the very real issues of heterogeneity and security. No widely held agreement exists in the industry regarding the basic assumptions that should be made about the communications networks underlying distributed systems. These issues will retard progress toward what could ultimately become a future of distributed computing based on objects. In the meantime, current methods for communication

among heterogeneous, distributed computers will continue to provide an important basis for cooperative processing; and, perhaps, object-oriented software techniques can improve our ability to specify, produce, use, and maintain the communications architectures, services, and protocols necessary for loosely-coupled, distributed computing.

In our paper, we presented a small experiment in applying object-oriented techniques to the specification of communication services. Specifically, we designed an abstract data type (ADT) for the Open Systems Interconnection (OSI) connection-oriented transport service (as defined in IS 8072). We then specified the ADT in Eiffel, an object-oriented programming language. Our experiment showed that such specifications are possible, though we identified some areas where natural language has advantages. Oddly enough, the IS 8072 definition included, purposefully, some ambiguous statements - ambiguity is anathema to ADTs.

One goal of object-oriented programming with ADTs is to specify the semantics and syntax of operations that a programmer can rely upon when using the services of an ADT. Here, we judged our transport service ADT too restrictive, and, so, we constructed a separate application programming interface (API) for the transport service. Our API exhibited increased flexibility when compared with our ADT, but the post-conditions in our API are probably impractical to evaluate at run-time. Perhaps, with more time and thought, some adaptation of our ADT could become a suitable API, but we are not convinced of that fact.

Our experiment with Eiffel and the OSI transport service encourages us that object-oriented techniques might improve our ability to specify, construct, use, and adapt communications architectures and protocols; but more thought and work is necessary before we can make such claims. We are convinced that ADTs can usefully represent unambiguous aspects of an OSI

service interface; however, we are unconvinced that ADTs (at least as represented in Eiffel) can adequately express the semantics of a peer-to-peer service that extends across two service interfaces.

## VI. References and Bibliography

- [1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, MA, 1986, 144 pages.
- [2] G. Agha and C. Hewitt, "Concurrent Programming Using Actors", in *Object Oriented Concurrent Programming*, The MIT Press: Cambridge, MA, 1987. pp. 37-53.
- [3] G. Agha and C. Hewitt, "Actors: A Conceptual Foundation for Concurrent Object Oriented Programming", in *Research Directions in Object Oriented Programming*, The MIT Press: Cambridge, MA, 1987, pp. 49-74.
- [4] G. Agha, "Foundational Issues in Concurrent Computing", *SIGPLAN NOTICES*, April 1989, pp. 60-65.
- [5] G. Agha, "Concurrent Object Oriented Programming", *Communications of the ACM*, September 1990, pp. 125-141.
- [6] J-M. Andreoli and R. Pareschi, "LO and Behold! Concurrent Structured Processes", *ECOOP/OOPSLA 90 Proceedings*, October 1990, pp. 44-56.
- [7] G. Andrews, *Concurrent Programming Principles and Practice*, Benjamin/Cummings Publishing Company: Redwood City, CA, 1991, 637 pages.
- [8] B. Anderson, "Fine-grained Parallelism in Ellie", *Journal of Object-Oriented Programming*, June 1992, pp. 55-61.
- [9] E. Aranow and T. Kehler, "Objects can set the stage: object tool suppliers targeting distributed system development: OMG's CORBA sets interoperability", *Software Magazine*, August 1992, pp. 13-14.

- [10] W. Athas and C. Seitz, "Multicomputers: Message-Passing Concurrent Computers", *COMPUTER*, August 1988, pp. 9-24.
- [11] S. Bailin, "An Object-Oriented Requirements Specification Method", *Communications of the ACM*, May 1989, pp. 608-623.
- [12] W. Bain, "Indexed, Global Objects for Distributed Memory Parallel Architectures", *SIGPLAN NOTICES*, April 1989, pp. 95-98.
- [13] A. Black, et al., "Distribution and Abstract Types in Emerald", *IEEE Transactions on Software Engineering*, January 1987, pp. 65-76.
- [14] J. van den Bos, "PROCOL: A Protocol Constrained Concurrent Object Oriented Language", *SIGPLAN NOTICES*, April 1989, pp. 149-151.
- [15] J-P. Briot, "Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment", *ECOOP 89 Proceedings of the Third European Conference on Object-Oriented Programming*, 1989, pp. 109-129.
- [16] J-P. Briot and J. Ratuld, "Design of a Distributed Implementation of ABCL/1", *SIGPLAN NOTICES*, April 1989, pp. 15-17.
- [17] J-P. Briot and A. Yonezawa, "Inheritance and Synchronization in Object Oriented Concurrent Programming", in *ABCL An Object Oriented Concurrent System*, The MIT Press: Cambridge, MA, 1990, pp. 107-118.
- [18] P. Buhr, et al., "Adding Concurrency to a Statically Type-Safe Object Oriented Programming Language", *SIGPLAN NOTICES*, April 1989, pp. 18-21.
- [19] D. Caromel, "A General Model For Concurrent and Distributed Object Oriented Programming", *SIGPLAN NOTICES*, April 1989, pp. 102-104.
- [20] G. Champine, et al., "Project Athena as a Distributed Computer System", *COMPUTER*, September 1990, pp. 40-51.
- [21] T. Christopher, "Message Driven Computing and its Relationship to ACTORS", *SIGPLAN NOTICES*, April 1989, pp. 76-78.

- [22] L. Crowl, "A Uniform Object Model for Parallel Programming", *SIGPLAN NOTICES*, April 1989, pp. 25-27.
- [23] W. Dally and A. Chien, "Object Oriented Concurrent Programming in CST", *SIGPLAN NOTICES*, April 1989, pp. 28-31.
- [24] N. Doi, et al., "An Implementation of an Operating System Kernel using Concurrent Object Oriented Language ABCL/C+", *ECOOP 88 Proceedings*, Springer-Verlag: NY, 1988, pp. 250-266.
- [25] J. Eliot, "Concurrency Features for the Trellis/Owl Language", *ECOOP 87 Proceedings*, Springer-Verlag: NY, 1987, pp. 171-180.
- [26] J. Faust and H. Levy, "The Performance of an Object Oriented Threads Package", *ECOOP/OOPSLA 90 Proceedings*, October 1990, pp. 278-288.
- [27] N. Gehani and W. Roome, "Concurrent C++: Concurrent Programming with Class(es)", John Wiley and Sons, 1988, pp. 1158-1178.
- [28] S. Habert, et al., "COOL: Kernel Support for Object Oriented Environments", *ECOOP/OOPSLA 90 Proceedings*, October 1990, pp. 269-277.
- [29] B. Hailpern and H. Ossher, "Extending Objects to Support Multiple Interfaces and Access Control", *IEEE Transactions on Software Engineering*, November 1990, pp. 1247-1257.
- [30] L. Heuser, et al., "Extensions to the Object Paradigm for the Development of Distributed Applications", *SIGPLAN NOTICES*, April 1989, pp. 111-113.
- [31] Y. Ichisugi and A. Yonezawa, "Exception Handling and Real-time Features in an Object Oriented Concurrent Language", in *Concurrency: Theory, Language, and Architecture*, Springer-Verlag: NY, 1991, pp. 92-109.
- [32] Y. Ishikawa, et al., "Object Oriented Real-Time Language Design: Constructs for Timing Constraints", *ECOOP/OOPSLA 90 Proceedings*, October 1990, pp. 289-298.
- [33] International Organization for Standardization, *Information Processing Systems - Open Systems*

*Interconnection - Transport Service Definition*,  
International Standard 8072, August 1984, 31 pages.

- [34] M. Jazayeri, "Objects for Distributed Systems", *SIGPLAN NOTICES*, April 1989, pp. 117-119.
- [35] M. Johnson, "Client/Server Computing: Recent Developments in OT indicate an evolution toward truly open systems", *Object Magazine*, October 1992, pp. 51-81.
- [36] D. Kafura and K. Lee, "Inheritance in Actor Based Concurrent Object-Oriented Languages", *ECOOP 89 Proceedings of the Third European Conference on Object-Oriented Programming*, 1989, pp. 131-145.
- [37] K. Kahn, et al., "Vulcan: Logical Concurrent Objects", in *Research Directions in Object Oriented Programming*, The MIT Press: Cambridge, MA, 1987, pp. 75-112.
- [38] K. Kahn and V. Saraswat, "Actors as a Special Case of Concurrent Constraint Programming", *ECOOP/OOPSLA 90 Proceedings*, October 1990, pp. 57-66.
- [39] G. Kaiser, et al., "MELDing Multiple Granularities of Parallelism", *ECOOP 89 Proceedings of the Third European Conference on Object-Oriented Programming*, 1989, pp. 147-166.
- [40] G. Kaiser, "Concurrent MELD", *SIGPLAN NOTICES*, April 1989, pp. 120-122.
- [41] G. Kaiser, "Transactions for Concurrent Object Oriented Programming Languages", *SIGPLAN NOTICES*, April 1989, pp. 136-138.
- [42] B. Kramer, "Specifying Concurrent Objects", *SIGPLAN NOTICES*, April 1989, pp. 162-164.
- [43] H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems", *OOPSLA 86 Proceedings*, October 1986, pp. 214-223.
- [44] H. Lieberman, "Concurrent Object Oriented Programming in Act 1", in *Object Oriented Concurrent Programming*, The MIT Press: Cambridge, MA, 1987. pp. 9-36.

- [45] J. Lim and R. Johnson, "The Heart of Object Oriented Concurrent Programming", *SIGPLAN NOTICES*, April 1989, pp. 165-167.
- [46] B. Liskov, "On Linguistic Support for Distributed Programs", *IEEE Transactions on Software Engineering*, May 1982, pp. 203-210.
- [47] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *ACM Transactions on Programming Languages and Systems*, July 1983, pp. 381-404.
- [48] S. Lujun, et al., "Concurrent Behaviors", *SIGPLAN NOTICES*, April 1989, pp. 168-170.
- [49] B. Martin, et al., "An Object-Based Taxonomy for Distributed Computing Systems", *COMPUTER*, August 1991, pp. 17-27.
- [50] G. Masini, et al., *Object Oriented Languages*, Academic Press: NY, 1991, 483 pages.
- [51] R. Mehrotra and J. Van Rosendale, "Concurrent Access in BLAZE 2", *SIGPLAN NOTICES*, April 1989, pp. 40-42.
- [52] B. Meyer, "Sequential and Concurrent Object-Oriented Programming", in *Eiffel A Collection*, Interactive Software Engineering, TR-EI-20/EC Version 2.3, 1990, pp. 151-163.
- [53] E. Moss, et al., "Panel Discussion: Object-Oriented Concurrency", *OOPSLA '87 Conference Proceedings*, October 1987, pp. 119-127.
- [54] C. Nascimento and J. Dollimore, "Behavior Maintenance of Migrating Objects in a Distributed Object Oriented Environment", *Journal of Object Oriented Programming*, September 1992, pp. 25-33.
- [55] O. Nierstrasz, "Active Objects in Hybrid", *OOPSLA 87 Proceedings*, October 1987, pp. 243-253.
- [56] O. Nierstrasz, "Two Models of Concurrent Objects", *SIGPLAN NOTICES*, April 1989, pp. 174-176.
- [57] O. Nierstrasz and M. Papathomas, "Viewing Objects as Patterns of Communicating Agents", *ECOOP/OOPSLA '90 Proceedings*, October 1990, pp. 38-43.

- [58] D. Rine, "Parallel and Distributed Processability of Objects", *Fundamental Informaticae XII*, 1989, pp. 317-356.
- [59] J. Rymer, "Common Object Request Broker: OMG's New Standard for Distributed Object Management", *Patricia Seybold's Network Monitor*, September 1991, pp. 3-28.
- [60] D. Shia, *The Manager Model - a Model for Distributed Computing*, DSET Corporation: New Jersey, 1992, 32 pages.
- [61] D. Shia, *Computing Environment on OSI (CEO) - a proposal*, DSET Corporation: New Jersey, March 5, 1992, 14 pages.
- [62] E. Shibayama, "How to Invent Distributed Implementation Schemes of an Object Oriented Concurrent Language", in *ABCL An Object Oriented Concurrent System*, The MIT Press: Cambridge, MA, 1990, pp. 87-105.
- [63] M. Singhal, "Deadlock Detection in Distributed Systems", *COMPUTER*, November 1989, pp. 37-47.
- [64] J. Stankovic, "Software Communication Mechanisms: Procedure Calls Versus Messages", *COMPUTER*, April 1982, pp. 19-25.
- [65] T. Takada and A. Yonezawa, "An Implementation of an Object Oriented Concurrent Programming Language in Distributed Enviroments", in *ABCL An Object Oriented Concurrent System*, The MIT Press: Cambridge, MA, 1990, pp. 133-155.
- [66] P. Wegner, "Granularity of Modules in Object-Based Concurrent Systems", *SIGPLAN NOTICES*, April 1989, pp. 46-49.
- [67] Y. Yokote, *The Design and Implementation of Concurrent Smalltalk*, World Scientific: New Jersey, 1990.
- [68] A. Yonezawa and M. Tokoro, "Object-Oriented Concurrent Programming: An Introduction", in *Object Oriented Concurrent Programming*, The MIT Press: Cambridge, MA, 1987. pp. 1-7.
- [69] A. Yonezawa, "Modelling and Programming in an Object Oriented Concurrent Language ABCL/1", in *Object*

*Oriented Concurrent Programming*, The MIT Press:  
Cambridge, MA, 1987. pp. 55-89.

- [70] A. Yonezawa, et al., "An Object Oriented Concurrent Computation Model ABCM/1 and its Description Language ABCL/1", in *ABCL An Object Oriented Concurrent System*, The MIT Press: Cambridge, MA, 1990, pp. 13-43.

*Appendix A. Eiffel Abstract Specification of IS 8072: The Connection-oriented Transport Service Definition*

```
deferred class TS_INTERFACE

    -- Transport Service Definition.

export T_CONNECT_request, T_CONNECT_indication,
        T_CONNECT_response, T_CONNECT_confirm,
        T_DATA_request, T_DATA_indication,
        T_EXPEDITED_DATA_request,
        T_EXPEDITED_DATA_indication,
        T_DISCONNECT_request, T_DISCONNECT_indication,
        TS_Interface, expedited_allowed

feature

    TS_Interface : TC_STATE;

    now : TIME;

    time_sent : TIME;

    TS_User_Invoked : DIS_REASON;

    expedited_allowed : BOOLEAN;

    cd_in, cd_out : CHANNEL[CTL_BLOCK[DATA]];

    dt_in, dt_out : CHANNEL[TSDU[DATA]];

    ed_in, ed_out : CHANNEL[E_TSDU[DATA]];

    T_CONNECT_request( Invoker : IDENTITY,
                       Called, Calling : ADDRESS,
                       Expedited_Option : BOOLEAN,
                       Quality_of_Service : QOS,
                       TS_User_Data : DATA
                     ) is

        require
            Invoker.isUSER;
            TS_Interface.isIdle;
            not Called.Void and not Calling.Void;
            not Quality_of_Service.Void;
            TS_User_Data.Void or else
```

```
(TS_User_Data.count > 0 and
TS_User_Data.count <=32);
```

**deferred**

**ensure**

```
cd_out.count = old cd_out.count + 1;
cd_out.last.isCR;
cd_out.last.address1 = Called;
cd_out.last.address2 = Calling;
cd_out.last.exp_opt = Expedited_Option;
expedited_allowed = Expedited_Option;
cd_out.last.q_o_s = Quality_of_Service;
cd_out.last.user_data = TS_User_Data;
```

```
TS_Interface.isOutgoing_Connection_Pending;
```

**end;** -- T\_CONNECT.request

```
T_CONNECT_indication(    Invoker : IDENTITY,
                          Called, Calling : ADDRESS,
                          Expedited_Option : BOOLEAN,
                          Quality_of_Service : QOS,
                          TS_User_Data : DATA
                        ) is
```

**require**

```
Invoker.isPROVIDER;
TS_Interface.isIdle;
cd_in.first.isCR;
Called = cd_in.first.address1;
Calling = cd_in.first.address2;
Expedited_Option = cd_in.first.exp_opt;
Quality_of_Service = cd_in.first.q_o_s;
TS_User_Data = cd_in.first.user_data;
```

**deferred**

**ensure**

```
cd_in.count = old cd_in.count - 1;
cd_in.empty or else
    old cd_in.first /= cd_in.first;
expedited_allowed = Expedited_Option;
TS_Interface.isIncoming_Connection_Pending;
```

**end;** -- T\_CONNECT.indication

```

T_CONNECT_response( Invoker : IDENTITY,
                    Responder : ADDRESS,
                    Expedited_Option : BOOLEAN,
                    Quality_of_Service : QOS,
                    TS_User_Data : DATA
                    ) is

    require
        Invoker.isUSER;

TS_Interface.isIncoming_Connection_Pending;
    not Responder.Void;
    Expedited_Option = false or else
    Expedited_Option =
        expedited_allowed;
    not Quality_of_Service.Void;
    TS_User_Data.Void or else
    (TS_User_Data.count > 0 and
TS_User_Data.count <=32);

    deferred

    ensure
        cd_out.count = old cd_out.count + 1;
        cd_out.last.isCC;
        cd_out.last.address1 = Responder;
        cd_out.last.exp_opt = Expedited_Option;
        expedited_allowed = Expedited_Option;
        cd_out.last.q_o_s = Quality_of_Service;
        cd_out.last.user_data = TS_User_Data;
        TS_Interface.isData_Transfer_Ready;

end; -- T_CONNECT.response

T_CONNECT_confirm( Invoker : IDENTITY,
                  Responder : ADDRESS,
                  Expedited_Option : BOOLEAN,
                  Quality_of_Service : QOS,
                  TS_User_DATA : DATA
                  ) is

    require
        Invoker.isPROVIDER;
        TS_Interface.isOutgoing_Connection_Pending;
        cd_in.first.isCC;
        Responder = cd_in.first.address1;
        Expedited_Option = cd_in.first.exp_opt;

```

```
Quality_of_Service = cd_in.first.q_o_s;  
TS_User_Data = cd_in.first.user_data;
```

**deferred**

**ensure**

```
cd_in.count = old cd_in.count - 1;  
cd_in.empty or else  
    old cd_in.first /= cd_in.first;  
expedited_allowed = Expedited_Option;  
TS_Interface.isData_Transfer_Ready;
```

**end;** -- T\_CONNECT.confirm

```
T_DATA_request(      Invoker : IDENTITY,  
                    TS_User_Data : DATA  
                    ) is
```

**require**

```
Invoker.isUSER;  
TS_Interface.isData_Transfer_Ready;  
not TS_User_Data.Void;  
TS_User_Data.count > 0;
```

**deferred**

**ensure**

```
TS_Interface.isData_Transfer_Ready;  
dt_out.seq = old dt_out.seq + 1;  
dt_out.last.content = TS_User_Data;  
dt_out.last.order = dt_out.seq;  
dt_out.last.time_stamp = time_sent;
```

**end;** -- T\_DATA.request

```
T_DATA_indication(      Invoker : IDENTITY,  
                        TS_User_Data : DATA  
                        ) is
```

**require**

```
Invoker.isPROVIDER;  
TS_Interface.isData_Transfer_Ready;  
not TS_User_Data.Void;  
TS_User_Data.count > 0;  
dt_in.first.order = dt_in.seq + 1;  
dt_in.first.content = TS_User_Data;  
dt_in.first.time_stamp < now;
```

**deferred**

```

    ensure
        TS_Interface.isData_Transfer_Ready;
        dt_in.seq = old dt_in.seq + 1;
        dt_in.count = old dt_in.count - 1;
        dt_in.empty or else
            dt_in.first /= old dt_in.first;

end; -- T_DATA.indication

T_EXPEDITED_DATA_request(      Invoker : IDENTITY,
                               TS_User_Data : DATA
                               ) is

    require
        expedited_allowed;
        Invoker.isUSER;
        TS_Interface.isData_Transfer_Ready;
        not TS_User_Data.Void;
        TS_User_Data.count > 0;
        TS_User_Data.count <= 16;

    deferred

    ensure
        TS_Interface.isData_Transfer_Ready;
        ed_out.seq = old ed_out.seq + 1;
        ed_out.last.content = TS_User_Data;
        ed_out.last.order = ed_out.seq;
        ed_out.last.time_stamp = time_sent;
        ed_out.last.before_dt = dt_out.seq + 1;

end; -- T_EXPEDITED_DATA.request

T_EXPEDITED_DATA_indication(  Invoker : IDENTITY,
                               TS_User_Data : DATA
                               ) is

    require
        expedited_allowed;
        Invoker.isPROVIDER;
        TS_Interface.isData_Transfer_Ready;
        not TS_User_Data.Void;
        TS_User_Data.count > 0;
        TS_User_Data.count <= 16;
        ed_in.first.order = ed_in.seq + 1;
        ed_in.first.content = TS_User_Data;
        ed_in.first.before_dt > dt_in.seq;
        ed_in.first.time_stamp < now;

```

```

deferred

ensure
    TS_Interface.isData_Transfer_Ready;
    ed_in.seq = old ed_in.seq + 1;
    ed_in.count = old ed_in.count - 1;
    ed_in.empty or else
        ed_in.first /= old ed_in.first;

end; -- T_EXPEDITED_DATA.indication

T_DISCONNECT_request(    Invoker : IDENTITY,
                        TS_User_Data : DATA
                        ) is

    require
        Invoker.isUSER;
        not TS_Interface.isIdle;
        TS_User_Data.Void or else
            (TS_User_Data.count > 0 and
             TS_User_Data.count <= 64);

deferred

ensure
    cd_in.empty;
    dt_in.empty;
    ed_in.empty;
    TS_Interface.isIdle;
    dt_out.count <= old dt_out.count;
    ed_out.count <= old ed_out.count;
    cd_out.count <= old cd_out.count + 1;
    cd_out.last.isDR;
    cd_out.last.user_data = TS_User_Data;
    cd_out.last.reason_code = TS_User_Invoked;

end; -- T_DISCONNECT.request

T_DISCONNECT_indication( Invoker : IDENTITY,
                        Reason : DIS_REASON,
                        TS_User_Data : DATA
                        ) is

    require
        not TS_Interface.isIdle;
        Invoker.isPROVIDER;
        cd_in.first.isDR;

```

```
TS_User_Data = cd_in.first.user_data;  
Reason = cd_in.first.reason_code;
```

**deferred**

**ensure**

```
TS_Interface.isIdle;  
cd_in.empty;  
dt_in.empty;  
ed_in.empty;  
cd_out.empty;  
dt_out.empty;  
ed_out.empty;
```

**end;** -- T\_DISCONNECT.indication

**end;** -- class *TS\_INTERFACE*

**deferred class** *TC\_STATE*

```
export isIdle,  
        isOutgoing_Connection_Pending,  
        isIncoming_Connection_Pending,  
        isData_Transfer_Ready
```

**feature**

isIdle :**BOOLEAN is deferred end;**

isOutgoing\_Connection\_Pending :**BOOLEAN is deferred end;**

isIncoming\_Connection\_Pending :**BOOLEAN is deferred end;**

isData\_Transfer\_Ready :**BOOLEAN is deferred end;**

**invariant**

```
        isIdle or else isOutgoing_Connection_Pending  
        or else isIncoming_Connection_Pending or else  
isData_Transfer_Ready;
```

**end;** -- class *TC\_STATE*

**deferred class** *IDENTITY*

```
export isUSER, isPROVIDER
```

**feature**

isUSER : **BOOLEAN is deferred end ;**

isPROVIDER : **BOOLEAN is deferred end ;**

**invariant**

isUSER **or else** isPROVIDER;

**end; --IDENTITY**

**deferred class CHANNEL[T]**

**export** first, last, count, empty, seq

**inherit TWO\_WAY\_LIST[T];**

**feature**

seq : *SEQ\_NUM*;

Zero : *SEQ\_NUM is deferred end ;*

**i nvariant**

seq >= Zero;

**end; --class CHANNEL**

**deferred class CTL\_BLOCK[T]**

-- A Control Block that can carry a transport connect  
request, connect  
-- confirm, or disconnect request.

**export** address1, address2, q\_o\_s, exp\_opt, reason\_code,  
isCR, isCC, isDR, user\_data

**inherit BI\_LINKABLE[T] rename item as user\_data;**

**feature**

isCR : **BOOLEAN is deferred end ;**

isCC : **BOOLEAN is deferred end ;**

```

    isDR : BOOLEAN is deferred end;

    address1, address2 : ADDRESS;

    q_o_s : QOS;

    exp_opt : BOOLEAN;

    reason_code : DIS_REASON;

end; -- class CTL_BLOCK

class TSDU[T]

    -- A Transport Service Data Unit (TSDU), the unit of data
    -- that a transport
    -- service user (TS_USER) sends on and receives from a
    -- transport connection
    -- This is the unit that the service is responsible for
    -- ensuring the integrity
    -- of.

export    order, time_stamp, content

inherit BI_LINKABLE[T] rename item as content;

feature

    order : SEQ_NUM;

    time_stamp : TIME;

end; -- class TSDU

class E_TSDU[T]

    -- This is an expedited transport service data unit, the
    -- unit of expedited
    -- data that the user submits to and receives from a
    -- transport connection.
    -- This is the unit of expedited data that the transport
    -- service is
    -- accountable for.

```

```

export      order, time_stamp, content, before_dt
inherit     TSDU[T];

feature

    before_dt : SEQ_NUM;

end; -- class E_TSDU

class DATA

export      count

inherit ARRAY[CHAR];

end; -- class DATA

class DIS_REASON

inherit INT;

end; -- DIS_REASON

class QOS

inherit INT;

end; -- class QOS

class SEQ_NUM

export infix "+", infix ">=", infix ">"

inherit INT;

end; -- class SEQUENCE_NUMBER

class TIME

export infix ">", infix "<", infix "<=", infix ">="

inherit     INT;

```

```
end; -- class TIME  
  
class ADDRESS  
  
inherit STRING;  
  
end; -- class ADDRESS
```

*Appendix B. Eiffel Abstract Specification of an Application Programming Interface to the Open Systems Interconnection (OSI) Transport Service*

**deferred class** *TS*

```
-- TRANSPORT_SERVICE
-- An Eiffel representation of ISO 8072 (the
international standard
-- connection-oriented transport service
specification). The
-- description given in here omits some of the
details of ISO 8072.
-- The purpose of this specification is to describe
the semantics
-- of the transport service interface in a form that
an application
-- programmer can invoke the services and comprehend
the semantics of
-- the service calls.
```

```
export listen, connect, disconnect, send, receive,
send_expedited, receive_expedited
```

**feature**

```
listen(user : TS_USER) is
```

```
-- A user invokes this feature to await a connection
--request from another transport service user.
```

**require**

```
valid(user);
user.isDisconnected;
```

**deferred**

**ensure**

```
user.isListening
or else
(user.isConnected and
other_user(user).isConnected);
```

```
end; -- listen
```

```
connect(user : TS_USER, remote_user : TS_USER) is
```

```

-- A transport service user invokes this feature to
  attempt
-- to actively connect to another transport service
  user.

  require
    valid(user);
    user.isDisconnected;
    valid(remote_user);
    remote_user.isListening
      or else
        remote_user.isDisconnected
      or else
        remote_user.isTrying;

  deferred

  ensure
    (user.isConnected and
     remote_user.isConnected)
      or else
        user.isDisconnected;

end; -- connect

disconnect(user : TS_USER) is

  -- A transport service user invokes this feature when
the
  -- user no longer wishes to send data on the
transport
  -- connection.

  require
    valid(user);
    user.isConnected;
    other_user(user).isConnected
      or else
        other_user(user).isDisconnecting;

  deferred

  ensure
    (user.isDisconnected and
     other_user(user).isDisconnected)
      or else
        user.isDisconnecting;

```

```

end; -- disconnect

send(user : TS_USER, tsdu : TSDU) is

-- The transport service user invokes this feature to
  cause
-- a transport service data unit to be sent to a
  remote user
-- on the transport connection

  require
    valid(user);
    not tsdu.Void;
    tsdu.content_sent.count > 0;
    user.isConnected;

  deferred

  ensure
    ((tsdu.order_sent =
      receive(other_user(user)).order_received)
     and
     (tsdu.time_sent <
      receive(other_user(user)).time_received)
     and
     (tsdu.content_sent =
      receive(other_user(user)).content_received))
     or else
     user.isDisconnected;

end; -- send

receive(user : TS_USER) : TSDU is

-- A transport service user invokes this function to
  receive
-- a transport service data unit from a remote user
  across a
-- transport connection.  If no transport service data
  units
-- are ready to receive, then the feature returns
  Void.

  require
    valid(user);
    user.isConnected or else user.isDisconnecting;

  deferred

```

```

    ensure
        Result.Void
        or else
            (Result.time_received > Result.time_sent
            and
            Result.order_received = Result.order_sent
            and
            Result.content_received =
                Result.content_sent)
        or else
            user.isDisconnected;

end; -- receive

send_expedited(user : TS_USER, etsdu : E_TSDU) is

-- A transport service user invokes this feature to
-- send an expedited
-- data unit across the transport connection.

require
    valid(user);
    not etsdu.Void;
    etsdu.content_sent.count > 0;
    etsdu.content_sent.count <= 16;
    user.isConnected;

deferred

ensure
    ((etsdu.order_sent =
receive_expedited(other_user(user)).order_received)
    and
    (etsdu.time_sent <
receive_expedited(other_user(user)).time_received)
    and
    (etsdu.content_sent =
receive_expedited(other_user(user)).content_received)
    and
    (user.tsdu_sent implies
    (etsdu.before_tsdu =
    user.last_tsdu_sent + 1)))

```

```

        or else
        user.isDisconnected;

end; -- send_expedited

receive_expedited(user : TS_USER) : E_TSDU is

-- A transport service user invokes this feature to
-- receive an expedited
-- data unit. If no expedited data unit is
-- available, then Void is
-- returned;

    require
        valid(user);
        user.isConnected
        or else
        user.isDisconnecting;

    deferred
    ensure
        Result.Void
        or else
        (Result.time_received > Result.time_sent
         and
         Result.order_received = Result.order_sent
         and
         Result.content_received =
             Result.content_sent
         and
         (user.tsdu_received implies
          Result.before_tsdu >
            user.last_tsdu_received))
        or else
        user.isDisconnected;

end; -- receive_expedited

valid(user : TS_USER) : BOOLEAN is

    deferred

end; -- valid

other_user(user : TS_USER) : TS_USER is

    require

```

```

        valid(user);

    deferred

        ensure
            Result /= user;
            valid(Result);

    end; -- other_user

end; -- class TRANSPORT_SERVICE

deferred class TS_USER

    -- This represents a user of the IS 8072
connection-oriented
    -- transport service.

    export isConnected, isListening, isDisconnecting,
            isDisconnected, isTrying, last_tsdu_received,
            last_tsdu_sent, tsdu_sent, tsdu_received, wakeup

    feature

        isTrying : BOOLEAN is
            deferred
            end; -- isTrying

        isConnected : BOOLEAN is
            deferred
            end; -- isConnected

        isListening : BOOLEAN is
            deferred
            end; -- isListening

        isDisconnecting : BOOLEAN is
            deferred
            end; -- isDisconnecting

        isDisconnected : BOOLEAN is
            deferred
            end; -- isDisconnected

        last_tsdu_received : SEQ_NUM;

        last_tsdu_sent : SEQ_NUM;

```

```

tsdu_sent : BOOLEAN is
    deferred
    end; -- tsdu_sent

tsdu_received : BOOLEAN is
    deferred
    end; -- tsdu_received

wakeup(remote_user : TS_USER) is

-- This feature is user by the transport service
-- provider to
-- indicate to the transport service user that some
-- event
-- has occurred that may require the transport
-- user's attention
-- This means: a remote user is trying to connect or
-- else
-- some data has arrived, or else some expedited
-- data has arrived.

    deferred
    end; -- wakeup

update_tsdu_sent is
    deferred

    ensure
        tsdu_sent = true;
        last_tsdu_sent = old last_tsdu_sent + 1;

    end; -- update_tsdu_sent

update_tsdu_received is
    deferred
    ensure
        tsdu_received = true;
        last_tsdu_received = old last_tsdu_received + 1;

    end; -- update_tsdu_received

Zero : SEQ_NUM is
    deferred
    end;

```

```

invariant
    last_tsdu_sent >= Zero;
    last_tsdu_received >= Zero;
    tsdu_sent implies (last_tsdu_sent > Zero);
    tsdu_received implies (last_tsdu_received > Zero);
    isDisconnected
        or else
        isListening
            or else
            isTrying
                or else
                isConnected
                    or else
                    isDisconnecting;

end; -- class TS_USER

```

```

class TSDU

```

```

    -- A Transport Service Data Unit (TSDU), the unit of
    data that a transport
    -- service user (TS_USER) sends on and receives from
    a transport connection
    .-- This is the unit that the service is responsible
    for ensuring the integrity
    -- of.

```

```

export order_sent, order_received, time_sent,
        time_received, content_sent, content_received

```

```

feature

```

```

    order_sent : SEQ_NUM;

    order_received : SEQ_NUM;

    time_sent : TIME;

    time_received : TIME;

    content_sent : DATA;

    content_received : DATA;

```

```

end; -- class TSDU

```

```

class E_TSDU

    -- This is an expedited transport service data unit, the unit of expedited
    -- data that the user submits to and receives from a transport connection.
    -- This is the unit of expedited data that the transport service is
    -- accountable for.

    export order_sent, order_received, time_sent, time_received,
             content_sent, content_received, before_tsdu

    inherit TSDU;

    feature

        before_tsdu : SEQ_NUM;

end; -- class E_TSDU

class DATA

    export count

    inherit ARRAY[CHAR];

end; -- class DATA

class SEQ_NUM

    export infix "+", infix ">=", infix ">"

    inherit INT;

end; -- class SEQUENCE_NUMBER

class TIME

    export infix ">", infix "<", infix "<=", infix ">="

    inherit INT;

end; -- class TIME

```