ACTOR SYSTEMS PLATFORM

DESIGN AND IMPLEMENTATION OF THE ACTOR PARADIGM IN A DISTRIBUTED OBJECT-ORIENTED ENVIRONMENT

by

Nandan Joshi

Project Report submitted to the Faculty of the

Virginia Polytechnic and State University

in partial fulfillment of the requirements for the degree of

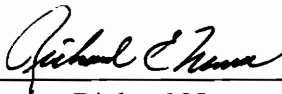
Master of Science

in

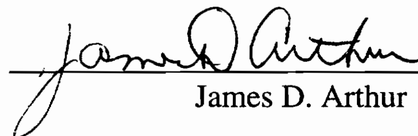
Computer Science

APPROVED

Dennis Kafura, Chairman



Richard Nance



James D. Arthur

August 1993
Blacksburg, Virginia

c.2

LD
5655
V851
1993
J673
c.2

ACTOR SYSTEMS PLATFORM
DESIGN AND IMPLEMENTATION OF THE ACTOR PARADIGM IN A DISTRIBUTED OBJECT-ORIENTED ENVIRONMENT

by

NANDAN JOSHI

Dr. Dennis Kafura, Chairman

Department of Computer Science, Virginia Tech

(ABSTRACT)

This project was undertaken as part of an effort to explore the design of object-oriented systems that are distributed, concurrent, real-time and/or embedded in nature. This work seeks to integrate the concurrency features of the actor model in a distributed, object-oriented environment, ESP. The integrated system, called the Actor Systems Platform (ASP), provides a platform for designing concurrent, distributed applications. The actor model provides a mechanism for expressing the inherent concurrency in an application. The concurrency in the application can be exploited by the distributed features available in ESP.

The actor abstraction in ASP is provided by a application-level class hierarchy in ESP. The message passing semantics of the actor model are implemented by using special operator overloading in C++. Cboxes are implemented to provide a synchronization mechanism and a means of returning replies. In a concurrent system, simultaneous

execution of an object's methods can cause its state to be inconsistent. This is prevented by providing a method locking mechanism using behavior sets.

While integrating the concurrency features of the actor model in an object-oriented environment, differences were encountered in determining the invocation semantics of the actor model and those of inherited methods. The problem is investigated and a taxonomy of solutions is presented.

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr.Kafura whose guidance has been vital for this project. Dr. Nance and Dr. Arthur have been very helpful and cooperative during the course of this work. Discussions with fellow students, especially Keung Lee, Greg Lavender and Horace Sequeira have been very useful.

I would like to thank my wife Milan for her cooperation and support. She has been very persistent in trying to get me to complete the project. My parents have always been a source of inspiration and moral support for me.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1. Background 1
1.2. Implementing Actors in ESP 2
1.3. Implementation Strategy 3
1.4. Organization of the Report 4
CHAPTER 2 MODELS AND LANGUAGES FOR CONCURRENCY 5
2.1. Introduction 5
2.2. Models of Concurrency 6
2.3. Concurrent languages 8
2.3.1. Actor-based languages 10
2.3.1.1. Act3 10
2.3.1.2. Cantor 11
2.3.1.3. Actalk 12
2.3.1.4. Actra 13
2.3.1.5. ABCL/1 14
2.3.1.6. ACT++ 14
2.3.2. Other Concurrent Object-Oriented Languages 15
2.3.2.1. ConcurrentSmalltalk 15
2.3.2.2. Hybrid 16
CHAPTER 3 A SURVEY OF DISTRIBUTED OBJECT-ORIENTED SYSTEMS 17
3.1. Introduction 17
3.2. Distributed Operating Systems 17
3.2.1. Communication Primitives 18

3.3. RPC and Futures 20

3.3.1. Remote Procedure Call 20

3.3.2. Futures 22

3.4. Object Based Systems 24

3.4.1. Introduction 24

3.4.2. Issues in the Design of Distributed Object-Based Systems 25

3.4.2.1. Object Structure 25

3.4.2.2. Object Composition 26

3.4.2.3. Object Interaction Management 27

3.4.2.3.1. Object Location 27

3.4.2.3.2. Method Invocation 29

3.5. Case Studies 31

3.5.1. Emerald 31

3.5.2. Avalon/C++ 34

3.5.3. Choices 36

CHAPTER 4 EXPERIMENTAL SYSTEMS PLATFORM 38

4.1. Introduction 38

4.2. Run-time Environment 39

4.2.1. Shadow process 39

4.2.2. ISSD 40

4.2.3. Mail Daemon 41

4.2.4. Node Kernels 41

4.2.4.1. Memory Management 41

4.2.4.2. Dynamic linking and loading 41

4.2.4.3. Message management: 42

4.2.4.4. Task Management: 43
4.3. Class Hierarchy in ESP 44
4.4. Futures 47
CHAPTER 5 THE ACTOR MODEL OF CONCURRENCY 51
5.1. Introduction 51
5.2. The Primitive Actor Model 51
5.2.1. Elements of the actor model 51
5.2.2. Message processing in actors 53
5.2.3. Continuations and customers 56
5.3. Modifications to the Primitive Actor Model 57
5.3.1. Granularity of actors 57
5.3.2. Reply messages and Cboxes 58
CHAPTER 6 DESIGN OF THE ACTOR SYSTEMS PLATFORM 61
6.1. Introduction 61
6.2. ASP : an application of ESP 61
6.3. Class Hierarchy in ASP 62
6.3.1. Class declarations 63
6.3.2. Actions performed in an actor system 67
6.3.2.1. Creating an actor 67
6.3.2.2. Send a message to another actor 68
6.3.3. Specify a replacement behavior 71
6.4. Cboxes: A Mechanism for Synchronization and Returning Replies 72
6.4.1. Implicit Vs Explicit Return 74
6.4.2. Design of Cboxes 74
6.5. Behavior Sets 78

6.5.1. Design of Behavior Sets 78
6.5.2. Implementation of Behavior Sets 81
6.6. Examples 83
6.6.1. Concurrent Factorial Computation 83
6.6.2. Dining Philosopher's Problem 87
CHAPTER 7 CONCLUSIONS 94
7.1. Introduction 94
7.2. Class Inheritance in ESP 95
7.3. Actor Model 96
7.4. The Problem of Invocation Semantics 97
7.4.1. Description of the Problem 97
7.4.2. A Taxonomy of Solutions 100
7.5. The Problem of Acquaintances 102
BIBLIOGRAPHY 104
APPENDIX A 108
APPENDIX B 123

LIST OF FIGURES

Figure 3.1 Flow of Execution in a Future-Based Remote Procedure Call........................23

Figure 3.2 Inheritance Hierarchy of Avalon/C++ Base Classes 35

Figure 4.1 The ESP Programming Environment..39

Figure 4.2 Mail Daemons and Node Kernels Connected to ISSD...................................40

Figure 4.3 Class Hierarchy in the ESP Kernel..44

Figure 4.4. Services Provided by Base Classes in the ESP Kernel..................................45

Figure 4.5 Dereferencing a Handle in ESP..45

Figure 4.6 Remote Method Invocation 47

Figure 5.1 An Actor...55

Figure 5.2 Continuations 56

Figure 6.1 Class Hierarchy in ASP..62

Figure 6.2 New Request Message Constructed 70

Figure 6.3 Message Passing Between Actor and Behavior 71

Figure 6.4(a) Implicit Return of Results Via Futures 73

Figure 6.4(b) Returning Results Explicitly Via Cboxes..73

Figure 6.5 Redirection of Messages to the Actor 73

Figure 6.6 Concurrent Factorial Example...84

Figure 6.7 Dining Philosophers Problem...88

Figure 7.1 Two Forms of Invocation 98

Figure 7.2 Taxonomy of Solutions for the Invocation Semantics Problem....................101

LIST OF EXAMPLES

Example 6.1 Actor Class 64

Example 6.2 Behavior Class 69

Example 6.3 Cbox Class 75

Example 6.4 Concurrent Factorial Example 85

Example 6.5 Dining Philosophers Problem 90

CHAPTER 1

INTRODUCTION

1.1. Background

The design and implementation of ASP (Actor Systems Platform) is part of ongoing research, the broader goal of which is to develop systems that are concurrent, distributed, real-time or embedded in nature. One of the issues under investigation is the effectiveness of concurrent object-oriented languages in implementing such systems. Object-oriented languages are of interest because they have the following properties. In an object-oriented language, real-world entities can be modeled as objects. Communication in an object-oriented language follows the message passing metaphor, which is analogous to the interaction between autonomous entities in a distributed system. By providing concurrency features in a object-oriented language, it is possible to model the distributed and concurrent aspects of the application domain.

The model of concurrency chosen was the actor model of computation [Agha 86a]. The actor model was chosen because it has several properties which are desirable in a concurrent, distributed system. Communication in the actor model is achieved by asynchronous message passing. Actors have the property of being dynamically reconfigurable [Agha 86b], i.e., communication paths between actors can be dynamically altered. This is a desirable property for systems which need to operate even when certain components or communication paths are unavailable. Several languages using the actor paradigm have been developed for distributed, parallel and real-time domains. The

results of the work done in the design of these languages has been encouraging. Chapter 2 will discuss some of these languages.

ACT++ is a concurrent language extension of C++. The design of ACT++ has been based on a modified version of the actor model proposed by [Kafura, Lee 89b]. The modifications to Agha's model will be discussed in detail in Chapter 5. These modifications enable development of the actor platform using imperative language constructs, while retaining the inherent concurrency of the actor model.

1.2. Implementing Actors in ESP

ESP [Leddy et. al. 88] is a distributed object-oriented system developed on top of UNIX. The focus of the present work is to integrate the concurrency of the actor model with the distributed and object-oriented features of ESP. The integrated system (called ASP) provides a test-bed for developing applications which are concurrent and distributed in nature. In ESP, distribution is achieved by creating objects remotely and sending to them requests for computation. The ESP kernel transparently manages the remote invocations. This provides an excellent platform for developing an actor system since the potential for concurrency in the actor model can be realized on such a platform. The actor model, on the other hand, provides the application designer with a powerful tool for expressing concurrency. Applications written in ASP can model the concurrent features of real-world entities by using the expressive power of the actor model. As explained before, ESP manages the distributed computation, thus enabling the application to exploit the inherent concurrency of the real-world system it is modeling. In summary, the expressive power of the actor model coupled with the distributed features of ESP provides a useful platform for developing concurrent, distributed applications.

In the course of this project, a problem was encountered in determining the invocation semantics while integrating actor-based concurrency and inheritance. [Kafura et. al. 90] argue that the problem is not specific to the actor model or ESP, but arises due to the deeper differences between the properties of concurrency and inheritance. The problem is described in detail in Chapter 7. A taxonomy of solutions for this problem is presented.

1.3. Implementation Strategy

ESP was developed to run on Sun3 workstations, the Symult s2010 multicomputer and the ESP proprietary hardware [Smith et. al. 89]. ASP has been implemented on a Sun3 workstation. Applications in ASP can run on a network of Sun3 workstations.

ASP has been developed with a modified version of the GNU C++ compiler, also used in developing ESP. The modification to the compiler (discussed in detail in Chapter 4) provides a special overloading of the pointer-to-member-function (→()) operator. Both ESP and the actor system take advantage of this special overloading to achieve the desired message passing semantics between objects.

The actor abstraction in ASP is provided by a layer of classes derived from the ESP base class. ASP runs as an application of ESP. The design of ASP does not require any modifications to the ESP kernel. This allows ASP to be easily portable between different versions of ESP. In addition, it is possible to run non-actor ESP applications on the same platform.

1.4. Organization of the Report

Concurrent languages and systems, the actor model and distributed systems are of interest to the present work. The review of literature in these areas is the subject of Chapters 2 and 3. Chapter 2 surveys several other models of concurrency including CSP [Hoare 78] and Milner's CCS [Milner 80]. Purely functional implementations of the actor model like Act3 [Theriault 83], actor-based concurrent languages like ABCL/1 [Yonezawa et. al. 87] and concurrent object-oriented languages like Hybrid [Papathomas 89b] are also studied. Languages like Hybrid, ABCL/1 and ACT++ [Kafura, Lee 89b] are of special interest since they are experiments in developing object-oriented concurrent languages. Chapter 3 discusses the design principles involved in the construction of distributed object-oriented languages. Examples of such systems, namely, Emerald [Black et al. 87], Avalon/C++ [Detlefs et. al. 88] and Choices [Campbell et al. 87] are presented. The design of ESP is studied in detail in Chapter 4, with special focus on the message passing and futures mechanisms. The actor model is presented in Chapter 5 along with a description of the modifications proposed by [Kafura, Lee 89b]. The design of ASP, which is the major focus of this work, is described in detail in Chapter 6. Of primary interest is the class hierarchy that was implemented to provide the actor abstraction and the design of Cboxes for returning replies to messages. The implementation of behavior sets to provide method locking is discussed. While integrating the concurrency features of the actor model with the object-oriented properties of ESP, the problem of invocation semantics was encountered. The problem and a taxonomy of solutions are presented in Chapter 7. The conclusion describes observations and experiences in the effort to provide concurrency in a distributed, object-oriented framework. The problem of maintaining acquaintance lists is discussed, which is a potential extension of this work.

CHAPTER 2

MODELS AND LANGUAGES FOR CONCURRENCY

2.1. Introduction

Concurrent systems and languages are of interest to language designers for several reasons:

- Concurrency is a natural way of expressing the interactions of real-world entities;
- If the underlying system can exploit the concurrency expressed by a language, it is possible to achieve a performance benefit over a purely sequential implementation.
- Replication of data and computing entities can improve the fault tolerance properties of the system.

This project is mainly concerned with the use of the actor paradigm to build concurrent systems. Models of concurrency and concurrent programming languages are of interest to this work. Several models of concurrency have been proposed. This chapter reviews some of these models, viz., CSP [Hoare 78], Milner's CCS [Milner 90], lambda-calculus, Petri-Nets [Molloy 82] and Data flow [Agarwal, Arvind 82]. The actor model of concurrency [Agha 86a] is discussed in detail in Chapter 5. Some concurrent languages designed with the actor paradigm, viz., Act3 [Theriault 83], Cantor [Athas, Boden 88], Actalk [Briot 89], Actra [McAffer, Berry 89], ABCL/1 [Yonezawa et. al. 87], ACT++ [Kafura, Lee 89b] are surveyed. A few concurrent languages based on other concurrency paradigms,

viz., Concurrent Smalltalk [Yokote, Tokoro 86] and Hybrid [Papathomas 89b] are surveyed as well.

2.2. Models of Concurrency

It is argued [Agha, Hewitt 87] that "the ability to model shared objects with changing local states, dynamic reconfigurability, and inherent parallelism are desirable properties of any model of concurrency". The actor paradigm [Agha 86a] has an object-based approach to computation. Actors are active entities that communicate with each other via message passing. Actors perform computation in response to messages received. The actions performed by an actor in response to a message are defined by the actor's script. Within a script, an actor may create a new actor. The existence of the newly created actor may be conveyed to other actors by sending them messages with an identifier to the new actor. Actor systems are dynamically reconfigurable, i.e., they possess the ability to create new objects and notify their existence to previously existing objects. In a system like CSP [Hoare 78], where the topology is fixed, it is not possible to meet this objective. Within its script, an actor may also specify a replacement behavior. The specification of a replacement behavior allows the actor to change its state.

Processes in CSP [Hoare 78] communicate via synchronous message passing. A synchronous message passing primitive implies a blocking send and a blocking receive. If a process A sends a message to process B, A is blocked until B accepts the message. When B does a receive, it is blocked until there is a message available. There is no necessity for the system to buffer messages. The communication patterns in CSP are pre-determined. [Agha 86a] argues that this violates the requirement of dynamic

reconfigurability. In an evolving system, it should be possible to create new objects and communicate their existence to previously existing ones.

[Milner 80] applied ideas in sequential programming to concurrent problems and found them to be insufficient. He decided that a fundamental approach, a new calculus, was needed to solve these problems. The Calculus of Communicating Systems (CCS) [Milner 80] was developed with the idea of indivisible interaction; interaction or communication between processes was central to the theory. CCS has the notion of observation equivalence of processes which can be expressed in an elegant algebraic notation. This describes processes which may differ in their internal behavior, but present identical external communication patterns. Communication in CCS (like CSP) is synchronous and resembles a handshake; a system based on the CCS model is not dynamically reconfigurable.

Functional programming, based on lambda-calculus, has been proposed as a concurrency model [Henderson 80]. The functional programming paradigm views the world as passive objects which are acted upon by functions that transform them. A functional program is composed of expressions, which are, in turn, composed of smaller subexpressions. Since assignment is not allowed in a purely functional language, it eliminates data dependency between subexpressions. Concurrency is achieved by eager evaluation, i.e., it is possible to concurrently execute all subexpressions in a functional program [Backus 78]. However, in a functional programming model, it is not trivial to model objects which change state, since it is not easy to represent the state of an object.

Petri Nets [Molloy 82] is a graphical technique used for modeling the states of systems. The states of a system are encoded as tokens marked on a graph. The graph structure illustrates the possible transitions of the system. The elements of a system modeled by a Petri Net are classified as either events or conditions. Events correspond to actions and conditions correspond to state descriptions of the system. For an event to occur, certain preconditions must be present. When the event occurs, the preconditions are removed and post-conditions are asserted. Petri Nets are very useful in proving properties such as mutual exclusion, absence of deadlock and reachability of a state in systems. Petri Nets do not have the computation power of a Turing machine.

Data flow [Agarwal, Arvind 82] extends the computational ability of Petri Nets to create an engine that is powerful enough to compute any computable function. Data flow associates information with the tokens and allows computation at the graph nodes. A transition in the Petri Net model becomes a Data flow object which consumes input and creates output. Concurrency in the Data flow model arises from multiple objects processing input simultaneously. There have been computer architectures designed with the Data flow approach and a language VAL developed to exploit Data flow computations.

2.3. Concurrent languages

[Agha, Hewitt 87] have identified the following as the most significant issues in the design of concurrent languages. The functional and object-oriented paradigms are contrasted with respect to their effectiveness in handling these issues.

(a) Shared Resources:

In a concurrent system, it must be possible to model objects that change state. It is not possible to model such objects in a purely functional system.

(b) Dynamic Reconfigurability:

The programming system must support creation of new objects. It should be able to convey the identity of such objects to previously existing objects. Systems based on CSP are not dynamically reconfigurable since the interconnection topology of processes in these systems is static.

(c) Inherent Parallelism

The concurrency available in a program should be deducible from the structure of the program. In a functional programming paradigm, assignment is not allowed. It is possible to concurrently execute all subexpressions in a program. The assignment statement in imperative languages creates dependencies in portions of the code. Without performing a flow analysis, it is not possible to determine the amount of concurrency available in the program.

In this section several concurrent languages are surveyed. These languages are broadly classified as:

(a) Languages based on the actor model;

(b) Languages based on other models of concurrency.

Languages based on the actor paradigm are further classified as:

1. Functional implementations (like Act3) which are more faithful to the actor model as proposed by [Agha 86a];

2. Languages like Actalk or ACT++ which attempt to integrate the actor model into imperative programming languages.

Some of the languages being surveyed, viz., Actalk [Briot 89], Actra [McAffer, Berry 89], ACT++ [Kafura, Lee 89b], support object-oriented features in addition to concurrency. The integration of these features is of interest for the following reasons. An object-oriented system treats the world as multiple, independent, interacting entities and hence lends itself to support concurrency features. In an object-oriented system, data and the operations that manipulate the data are typically stored together. This property of referential locality can improve the performance of a concurrent system by reducing the amount of communication activity. However, certain concurrency and object-oriented features interfere with each other making it difficult to integrate the two. For example, concurrent execution of an object's methods can compromise the encapsulation provided by object-oriented languages. Other conflicts have also been observed by [Kafura et. al. 90] and [Papathomas 89a].

2.3.1. Actor-based languages

2.3.1.1. Act3

Act3 [Theriault 83; Agha 86b] is an actor-based programming language developed for the Apiary architecture. Apiary is a parallel architecture based on a network of Lisp machines. Act3 is written in a Lisp-based language called Scripter and has the flavor of a functional programming language. The implementation is faithful to the actor model as proposed by [Agha 86a].

An Act3 program consists of behavior definitions and commands to create actors and send communications to them. A behavior definition consists of an identifier for the actor, a list of acquaintances to whom communications may be sent and a script which defines how the actor will respond to the communication it receives. All commands in the script of an actor can be executed in parallel.

Whenever an actor receives a message, it is pattern-matched with the communication handlers in the actor and dispatched to the appropriate handler. No assignment is allowed, hence all state changes are accomplished by the become operation which specifies the replacement behavior. The become operation also serves as a synchronization construct.

2.3.1.2. Cantor

Cantor [Athas, Boden 88] was developed as a platform for expressing concurrent computation on fine-grain multicomputers. A typical architecture for a multicomputer would be a collection of thousands of small processor chips, each with its own instruction processor, memory and a message passing interface. The nature of the architecture, specifically the limited addressable memory, precludes usage of programming models developed for sequential architectures. The actor model of computation is well suited for this type of architecture. The programmer can express computation by building distributed data structures that are collections of objects. The compiler and the runtime system share the burden of distributing and managing these objects across the nodes of the multicomputer. Other issues of interest are the allocation of resources like storage, load balancing on the communication links and work load balancing on the instruction processors.

2.3.1.3. Actalk

Actalk [Briot 89] is a system designed for classifying and simulating actor languages. The Actalk kernel was built by introducing actors into the Smalltalk-80 language. The integration of actors into the language converts the computation model based on passive, sequential objects communicating via synchronous message passing into active concurrent entities communicating via asynchronous message passing. The user interfaces with the system via two classes: Actor and ActorBehavior. The Actor class defines the mail queue that buffers the messages. The ActorBehavior class defines the script of the actor, i.e., the response of the actor to its messages. To maintain compatibility with the Smalltalk-80 language, Actalk supports the assignment operation. Since state changes can occur while processing a message, Actalk supports only actors which process messages in a serialized manner.

The main objective in Actalk is to implement actors over a standard environment such as Smalltalk-80. Another objective of the Actalk project was to model some representative object-oriented concurrent programming models and languages based on the actor concept. [Briot 88] describes the extensions of the basic Actalk kernel to simulate:

(a) the actor model of computation as proposed by Agha; and

(b) the actor model as implemented in ABCL/1.

Modifications to the Actalk kernel to simulate these languages will be described. The actor model of computation as described by Agha postulates that all state changes in an actor be modeled by specifying a replacement behavior. Every behavior of an actor accepts only a single message and specifies its replacement behavior. Hence messages queued in an actor's mailbox are consumed by subsequent behaviors of the actor. Since

assignment is not allowed, it is possible for these behaviors to execute concurrently. To model the above abstraction, the designers of Actalk created a subclass of their ActorBehavior class called AghaActorBehavior. This class provides the **replace** method to specify replacement behavior and also modifies the manner in which messages in the actor's queue are handled.

2.3.1.4. Actra

The objective of the Actra project [McAffer, Berry 89] is to provide an object-oriented programming environment for developing industrial computing systems. The developers of Actra are strongly influenced by the actor model but do not believe that industrial applications require a massively parallel architecture. Hence, the system has been designed to operate on conventional shared memory architectures.

Actra provides the actor abstraction in a Smalltalk-80 environment by modeling actors as a Smalltalk class. To support communication between actors, the developers of Actra have modified the Smalltalk-80 environment to support the process structuring primitives of Harmony [Gentlemen 85]. Harmony is a real-time object-based kernel which supports multi-tasking and multi-processing in a shared memory environment.

Although the developers of Actra claim to be influenced by the actor model, the computation model of Actra differs from the classic actor model in several ways. In Actra, active objects coexist with Smalltalk processes (or co-routines) which are not first-class objects. Active objects communicate by synchronous message passing. Hence, an object which sends a message to another object is blocked until the result of the

computation is returned. Actra does not provide any facilities for specifying replacement behavior.

2.3.1.5. ABCL/1

ABCL/1 [Yonezawa et. al. 87] supports the actor abstraction, but with certain variations. The actor model considers all elements as active objects, but ABCL/1 differentiates between active and passive objects. Active objects communicate by message passing, whereas passive objects communicate via function calls.

The communication mechanism provides for three types of message passing: past, now and future. The past message is simply an asynchronous message send where the sender does not expect a result. The now message is similar to a blocking call by an object. The future message permits delayed evaluation of a request message via a future variable. All message passing is asynchronous, with the exception of messages to the self object. This avoids the possible deadlock condition of an object invoking its own behavior and waiting on a reply. ABCL/1 does not provide the become facility to alter an object's behavior. Control of message acceptance is achieved by means of a select operator. Depending on where the object is executing in its select script, it may respond to only specific messages. ABCL/1 has no built-in system of inheritance.

2.3.1.6. ACT++

ACT++ [Kafura, Lee 89b] is a concurrent object-oriented language based on the actor model proposed by [Agha 86a]. The language is a hybrid of the actor model and C++. The designers of ACT++ felt the necessity of integrating the actor model into a more

conventional language like C++. Requirements of the application domain necessitated the deviation from Agha's proposal for the actor model.

The effectiveness of fine-grained instruction-level concurrency in Agha's actor model is based on the availability of a large number of processors with extremely fast inter-process communication. [Kafura, Lee 89b] argue that instruction-level concurrency will not be appropriate in the application domain of real-time systems. A behavior script in ACT++ is an ultra-light process, i.e., it is on the order of a procedure. Concurrency in ACT++ is obtained by:

- specifying a replacement behavior (intra-object concurrency); or
- communication among autonomous existing actors (inter-object concurrency).

Another deviation from Agha's actor model is the use of Cboxes. A Cbox (named after the structure in Concurrent Smalltalk) is used as a mechanism by which an actor can receive a reply message from the server of a request message. The requesting actor does not have to create continuations to wait for the reply.

2.3.2. Other Concurrent Object-Oriented Languages

2.3.2.1. ConcurrentSmalltalk

ConcurrentSmalltalk [Yokote, Tokoro 86] extends Smalltalk-80 by adding concurrency features. Objects are classified into ordinary objects which have multiple concurrent activations and atomic objects which are serialized resources. The developers of ConcurrentSmalltalk have also observed the interference between concurrency and inheritance. Messages sent to self and super are interpreted as local procedure calls as

opposed to messages sent to other objects. If messages to oneself were not handled in the above manner, it would result in deadlock. Since an object can have multiple concurrent activations, ConcurrentSmalltalk provides secretaries to serialize the execution of ordinary objects.

2.3.2.2. Hybrid

Hybrid [Papathomas 89b] is the result of a project to develop an integrated concurrent, distributed, object-oriented language and run-time system. Objects in Hybrid are multi-threaded entities. A thread can be in either one of the following states: idle (no processing), running (processing a request) or blocked (waiting for a reply). The types of messages allowed are termed reflexes (asynchronous messages without an associated reply) and delegated calls (asynchronous messages expecting a reply). When an object makes a delegated call, the thread is left in an idle state waiting for a reply. In this state, the object can switch to another thread and respond to other messages. When the reply for the delegated call is received, the object can resume execution of the blocked thread. Objects are classified as local (contained within the scope of another object) or remote (independent object). Synchronization is done only for remote objects.

CHAPTER 3

A SURVEY OF DISTRIBUTED OBJECT-ORIENTED SYSTEMS

3.1. Introduction

This project is concerned with the integration of the actor model of concurrency with a distributed object-based system, ESP. Of interest is the design of distributed systems, in particular, object-based distributed systems. This chapter will explore design issues relevant to the construction of distributed object-based systems. The RPC communication and coordination primitives in ESP (futures) are critical factors in the design of the actor system. The RPC and futures communication primitives are discussed in detail. Examples of distributed object-based systems, viz., Emerald [Black et. al. 87], Avalon/C++ [Detlefs et. al. 88] and Choices [Campbell et. al. 87] are reviewed.

3.2. Distributed Operating Systems

[Tanenbaum, Renesse 85] define a distributed operating system as "one that looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs)". The central idea in the above definition is transparency; the operating system should present itself to the user as a single processor, not as a collection of processors. An example of the latter is a network operating system, a collection of personal computers each running its own individual operating system, tied together by a local network.

To clarify the concept of a distributed operating system, it is compared to a network operating system. A network operating system differs from a distributed operating system in the following ways [Tanenbaum, Renesse 85]. First, to access a machine in a network system, the user has to explicitly log into the machine, i.e., the user has to be aware of the physical address or name of the machine. A distributed system provides the users with the appearance of a single machine. Second, files on remote machines in a network operating system can be accessed via the remote login session or by transferring the files to the local machine. This implies that the user has to be aware of the location of the files. In a distributed operating system (or even a distributed file server), the users can access remote files as if the files were resident on the same machine. Third, a network operating system provides no fault tolerance, i.e., if a machine on the system becomes unavailable for any reason, users accessing that machine cannot continue using the system. In contrast, if a node in a distributed system goes down, users of the system can access many or even all of the services (if service replication is supported), with some performance penalty.

The following section is devoted to the discussion of relevant issues in the design of interprocess communication mechanisms in distributed operating systems. RPC is contrasted with futures, as a mechanism for interprocess communication. RPC is of interest since it has been studied extensively and is used in many distributed systems. ESP uses futures as a means of accomplishing asynchronous inter-object communication.

3.2.1. Communication Primitives

The client-server model is a popular model of communication in the design of distributed systems. A client that wants a service sends a request to the server and waits for a reply.

In its most basic form, two primitives SEND and RECEIVE are provided. The SEND primitive specifies the message and its recipient, while the RECEIVE primitive specifies the sender and a repository for the message (like a mailbox). Several choices have to be made regarding the semantics of these primitives:

(a) Reliable versus Unreliable Primitives

On one end of the spectrum, the SEND operation could put the message on the network and not concern itself about the delivery of the message. On the other end, the SEND could guarantee delivery by acknowledgements and retransmissions, if necessary. The SEND operation in ESP (refer Chapter 4) adopts the former policy. Guarantee of message delivery is delegated to the network layer protocol, TCP/IP running on a network of Sun workstations.

(b) Blocking versus Non-blocking Primitives:

In a non-blocking SEND operation control is immediately returned to the sender of the message after the message has been queued for transmission. In the blocking case, the sender is blocked until the message is sent or, in the case of a reliable communication model, an acknowledgement has been received for that message. The benefit of implementing non-blocking primitives is that they can provide some potential for parallelism, i.e., the process issuing the send can perform computation while t

he message is traveling over the network. On the other hand, non-blocking primitives may be undesirable since they result in irreproducible and thus, hard to debug, programs.

(c) Buffered vs. Unbuffered Primitives:

Another decision that has to be made is whether messages should be buffered or not at the receiver. In a CSP style rendezvous, when a sender has a message for a receiver that has not done a RECEIVE as yet, the sender is blocked until the receiver is ready to receive. In this style of messaging, no buffering of messages is done. If buffering is desired, the senders of requests are provided with a mailbox identifier to which they can send messages. In this manner, multiple requests to the receiver can be queued until the receiver is ready to process them. Buffering raises issues like buffer management, protection and special treatment of high-priority messages. This last issue arises in languages like ABCL/1, a concurrent object-oriented language which supports express mode messages. These messages can interrupt processing of a message that was sent in ordinary mode, thus allowing interrupt processing for objects.

3.3. RPC and Futures

3.3.1. Remote Procedure Call

The RPC (remote procedure call) mechanism exploits the similarity between a client waiting on a server to process its request and a function call. This mechanism was found to be attractive since the semantics of procedure calls was well understood by application developers. RPC has been widely accepted as a mechanism for interprocess communication in distributed systems.

The RPC style of communication is described as follows. When a client C wishes to invoke a method p of a remote server S, it makes a normal function call:

result = S.p (arg1, arg2, ... , argn)

The call invokes a stub procedure, usually generated by a pre-processor and linked to the application. The stub procedure may be statically linked at compile-time or dynamically linked upon the call to function p. The stub procedure packages the parameters into a message of a standard format like

send (S, p, arg1, arg2, ... , argn)

and sends it to the remote machine. The stub procedure blocks until it receives a reply. On the receiving machine, another stub should be waiting for the message. When it receives the message, it unpackages it and makes a local call to the server with the arguments from the message. The returned result follows the same path in the opposite direction.

RPC hides the details of locating a server on a distributed network from the programmer. The programmer simply makes a function call to access a certain service in the distributed system; he is not aware of the location of the service, or whether it is available locally or at a remote location. The details of locating the physical server are abstracted from the client. It is not possible for the client application to route a request to a specific machine in the network. The RPC mechanism locates the required service by using a global name server.

The semantics of RPC do not lend itself very well to applications involving broadcasting and multicasting. Consider a scenario in which a sender wishes to simply ascertain the existence of a mail service. All the sender cares about is whether there is a mail server running on the network at that instant. It would like to broadcast its message to all servers

and block, waiting for the first reply to arrive. RPC does not lend itself to applications which require sending multiple messages and waiting on some or all of the replies to those messages.

3.3.2. Futures

[Walker et. al. 90] have designed and implemented an "asynchronous execution facility, futures, that addresses the inadequacies of RPC, while retaining the benefits that make RPC so useful". They criticize the RPC communication model for the following reasons. Due of the synchronous nature of RPC, it is difficult to exploit the inherent parallelism of a distributed system. This leads to poorer performance than mechanisms such as explicit message passing. In addition, as explained in the previous paragraph, RPC is ineffective in managing multiple remote computations. In contrast, futures, while maintaining the call syntax of RPC, allow the caller and callee to execute in parallel. Only the client application makes references to futures, hence existing server implementations can be used in conjunction with futures.

A future-based communication works in the following manner. Whenever a client initiates a remote computation (Figure 3.1), an entity (future) is generated. An identifier for the future is associated with the invocation and passed along to the server. The kernel at the server end receives the identifier and stores it. When the server's computation completes, its results are returned to the client along with the future identifier. If the client attempts to evaluate the future identifier (and thereby obtain the results of the computation) before the results are returned, it is blocked until the results arrive. On the other hand, if the results are available at the time the client evaluates the future identifier, it will not block. The future identifier serves to associate the results with the request for

computation. While the server is processing the request, the client can perform other computation, thus exploiting the inherent parallelism of the distributed system.

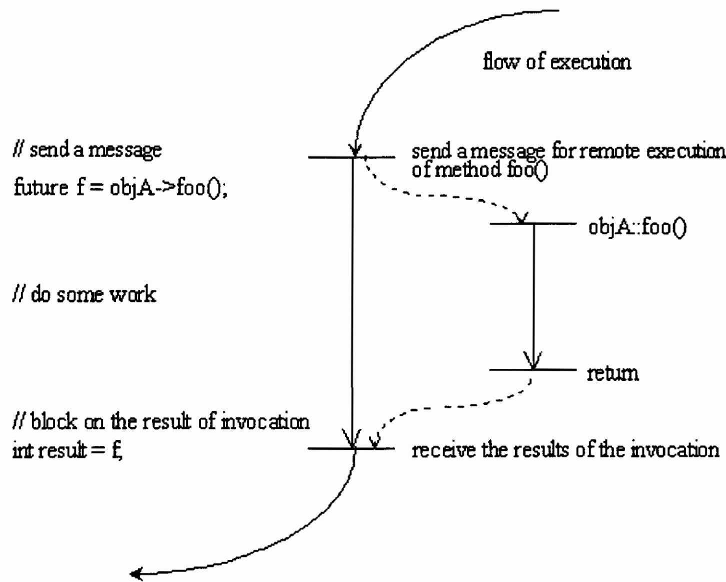


Fig 3.1. Flow of execution in a future-based remote procedure call

Multiple futures can be grouped into futuresets [Walker et. al. 90]. Such collections of futures can be addressed as a single entity. Futuresets are especially useful in applications where multiple results are returned, for example, in a multicasting operation. Consider the case of an application wishing to wait on the first of a number of futures. Since the application cannot predict the order of arrival of the futures, it cannot know which future should be evaluated first. Futuresets solve this problem by allowing the sender to wait on the futureset.

3.4. Object Based Systems

3.4.1. Introduction

[Wegner 87] defines a object-based language as one which supports objects as a language feature. For a language to be object-oriented, it must also support object classes and class inheritance. A class is defined as a template from which objects can be created. All objects belonging to a class have common operations and exhibit identical behavior. Inheritance is defined as the property by which classes can inherit operations from their superclasses and have their operations inherited by their subclasses [Wegner 87]. Of systems being studied in this section, Emerald [Black et. al. 87] is not object-oriented since it has no support for inheritance. On the other hand, Choices [Campbell et. al. 87] and ESP [Smith 88] are examples of object-oriented systems. The term object-based systems is used, in general, to refer to all of these systems.

Object-based languages encourage the design of systems in which multiple autonomous entities perform computations by interacting with each other. Distributed systems permit multiple independent systems connected by a network to be thought of as a single identity. The combination of these principles define a system in which a set of autonomous modules perform computation distributed over multiple, loosely coupled processors. An object-based distributed programming system (DOBPS) inherits the characteristics of both object-based systems and a decentralized computing environment. These systems are characterised by the following properties [Chin, Chanson 91]:

- Distribution: Computing is performed by a network of independent, possibly heterogeneous workstations.

- Transparency: The details of the distributed environment are hidden from the user. System services and objects may be invoked without knowledge of their location in the system.
- Concurrency: It should be possible to concurrently execute objects in an application on distinct processors.

3.4.2. Issues in the Design of Distributed Object-Based Systems

This section discusses issues related to the design of distributed object-based systems (DOBPS) [Chin, Chanson 91] .

3.4.2.1. Object Structure

This sub-section discusses properties relating to object structure such as object granularity and object composition.

[Chin, Chanson 91] define the granularity of an object as "the relative size, overhead and amount of processing performed by the object". Several models of object granularity have been proposed. One extreme is the model in which only large-grain objects such as files and single-user databases are supported. Object interactions are minimal, but so is the potential for inter-object concurrency. Designers of a DOBPS may choose to provide support for medium-grained objects as well, viz., queues, linked lists. In this case, synchronization of objects is done at the level of the medium-grained objects and there is a high potential for concurrency among these objects. A consistent data model is not provided in either of the two object models discussed above, since fine-grain objects such as basic data types are represented as conventional programming language abstractions. To provide a uniform object model for computation, it is necessary to support fine-grain

objects, in addition to medium-grain and large-grain objects. Fine-grain objects such as basic data types are usually contained within medium-grain objects. Fine-grain objects typically have a large number of interactions with other objects. For example, integers are represented as objects; therefore the computation 2+2 causes a message + to be sent to an instance of class integer. The message traffic in a system implementing fine-grained objects is high.

3.4.2.2. Object Composition

Composition of objects can be defined as the relationship between the processes and objects in a distributed object-based system. A DOBPS can be composed of processes which are bound to the objects in the system only at invocation time (passive object model) or the processes may be bound to the objects, inside of which they execute, at all times (active object model). The passive and active object models will be discussed in this section.

In the passive object model, a single process can, in the course of a computation execute within several different objects. When a process switches between objects, it has to map into the address space of the new object. The process is bound to a given object only during the time it is executing within the object. The advantage of the passive object model is that several processes may be bound to a single object, thereby increasing the potential for concurrency. However, mapping a process into the address space of multiple objects can be expensive.

The active object model conforms to the client-server model of computing. When an object is created, one or more processes are created for the object. Requests to the object

are satisfied by these processes. When the object is destroyed, so are its associated processes. ESP follows the active object model; a single process is created for each object.

3.4.2.3. Object Interaction Management

An important function of a distributed object-based system is managing invocations between objects, i.e., locating the called object, invoking its methods and returning the results to the caller. The following design issues are discussed:

1. Object location;
2. Method invocation;
3. Detection of invocation failures.

3.4.2.3.1. Object Location

One of the goals of designing an object location strategy is to provide transparent access to all objects. Clients of an object should not have to be aware of the location of the object in the system. When a client makes an invocation, the system examines the object identifier and determines where the object is located in the system. The object identifier should be unique across the system. In addition, the location strategy should allow objects to migrate across the system.

One scheme requires encoding the location of the object within its identifier. To locate an object, the system examines the appropriate field of the identifier. This scheme is simple but restricts object mobility; since the location of the object is encoded within the identifier, the object cannot change location without modifying its identifier. Modifying

the object identifier would imply having to inform all objects in the system which would reference the migrating object.

Another scheme uses a distributed name server to maintain global information about the location of objects in the system. Certain workstations in the system can be designated as name servers. Name servers cooperate with each other to maintain the most recent information about objects in the system. Requests for locating objects in the system may be directed to any of these servers. Each machine maintains only partial information about objects in the system. If a server cannot service a location request, it delegates that request to another server. Whenever a new object is created in the system, the information in each (relevant) name server has to be updated. The same is true in the event an object moves across machines. The disadvantage of this scheme is that it is difficult to maintain consistent information across all name servers in the system, since updating the name servers is not an instantaneous operation.

The object location scheme used by ESP is similar to the first scheme discussed in this section. ESP uses a system-wide unique structure (handle) to identify objects in the system. A handle is defined as:

```
struct   handle
{
    node node_id;     // identifier for the node
    appl appl_id :8;  //entry in the application table for this node
    clas clas_id :8;  //entry in the class table for the application
    inst inst_id :16; // entry in the instance table for this class
};
```

At a given node, an object is located as follows. The entry **appl_id** in the application table is used to identify the class table for the object. The entry **clas_id** in the class table

is then used to identify the instance table for the object. The entry **inst_id** in the instance table points to the object identified by the handle. In this scheme, the node at which the object is located is part of the structure. Hence, this scheme suffers from the same drawbacks as the first scheme.

3.4.2.3.2. Method Invocation

When the object is located in the system, it is the responsibility of the system to invoke its method and return the results to the caller. The strategy for handling method invocation is dependent on the object model used by the system. The following schemes are discussed :

(a) Message Passing

Distributed systems based on an active object model use message passing to implement method invocations. In ESP, kernels at the sending and receiving ends perform the functions related to the delivery of the request message and the return of results, respectively. Wehn an object invokes a method of another object, a message is passed from the node of the invoking object to the node of the object whose method is invoked. The kernel at the receiving end processes the request message and invokes the method of the appropriate object.

The return of results can be performed in two different ways. The server object can explicitly send a reply message to return the results back to the client. In ESP, the reply is implicit; when the method invocation is complete, the kernel formats the reply message and sends it to the client object. The kernel at the node of the object receiving the results

processes the reply message and forwards the results to the object. The issue of return semantics will be discussed in greater detail in Chapter 4.

In a message passing scenario, no explicit connection between the caller and called object has to be established. Rather the binding between the client and server is dynamic, effected whenever an invocation takes place.

(b) Direct Invocation

Systems that provide a passive object model support the direct invocation scheme for handling method invocations. In a passive object environment, a single process is responsible for accomplishing all the tasks in a computation. In the event of an invocation, the process switches from the caller object to the called object to execute the method invoked. When the invocation is local, the following steps are taken to accomplish the invocation:

- The state of the process and the current object are saved;
- Invocation parameters are loaded onto the stack;
- The called object is loaded into memory and a procedure call is made to execute the method;
- When the execution is complete, the results are returned to the caller object and the state of the process is restored.

In the event the invocation is remote, a message containing the invocation parameters is formatted and sent to the workstation on which the remote object resides. A worker

process is created at the remote site to execute the method invoked. On completion, the results are formatted into a message and sent back to the workstation where the caller resides. The worker process created on the remote workstation is terminated.

Method invocation in a direct invocation scheme resembles a procedure call for a local invocation and a remote procedure call for a remote invocation. In the case of a local invocation, the direct invocation scheme is more efficient, since it does not generate a message send.

[Chin, Chanson 91] conclude that development of distributed systems is simplified by the use of objects, since "objects serve well as the units for protection, recovery, security, synchronization and mobility". Using the object abstraction creates a common primitive that can be manipulated by both the user as well as the system, thus enabling users to express their ideas more conveniently.

3.5. Case Studies

A few distributed object-based systems are discussed, with emphasis on the design issues presented in the previous sections. ESP [Smith 88] is discussed in greater detail in Chapter 4.

3.5.1. Emerald

Emerald [Black et. al. 87] is an object-based language (it does not support inheritance) for programming distributed systems and applications. The designers of Emerald intended to provide support for distribution in the language instead of only in the operating system. This distinguishes Emerald from similar efforts, where distribution is provided as an

operating system facility without language support. Since the language provides support for distribution, the programmer is not responsible for tasks such as locating the communications target, packaging parameters and so on.

The goals of the Emerald project are summarized as follows:

- the ability to define local and distributed objects;
- efficient intra-node and inter-node communication between objects;
- capacity to exploit the parallelism inherent in a distributed system.

All information in Emerald is represented as objects. Regardless of the representation, all objects are defined using a consistent model. The same model holds for defining small objects such as integers and large objects like directories. Every object has the following four components: a unique identifier, internal data representation, set of operations and an optional process. In Emerald, objects are classified as active or passive, depending on whether or not they have a process associated with them. Furthermore, each object has an attribute that specifies the objects location, i.e., the node on which the object resides. This is necessary, since, in Emerald, objects can move at any time. However, when an object's method is invoked, the caller is not concerned with the current location of the object. The issue of object mobility is discussed later in this section, with respect to method invocation.

Emerald supports both inter-object and intra-object concurrency. Intra-object concurrency arises because multiple concurrent invocations of an object's methods are permitted. This is in contrast to ESP where the system permits only a single operation of an object to be

invoked at any time. The local ESP kernel guards access to an object's operations and thereby restricts multiple invocations of an object's methods.

Emerald differs from languages such as Smalltalk in its support for types. The designers of Emerald decided to provide support for types for the following reasons:

(a) In Smalltalk, when an invocation is attempted on an object that does not support the operation, the error is detected at run-time. Since objects in Emerald are typed, such errors can be detected at compile-time.

(b) Type errors are detected statically and at run-time there is no possibility of type collisions occurring. [Black et. al. 87] claim that this has permitted the designers of Emerald to implement a more efficient method lookup strategy than Smalltalk.

Emerald adopts a dual view of object location. Programmers may choose to be aware of the location of an object or choose to ignore it. In a distributed object-based system, an object should be able to access another object without concern for the object's location. In Emerald, since the system is responsible for locating objects, objects may be invoked without regard for their location. On the other hand, a programmer can exploit the location characteristics of objects to reduce communication overheads. For example, objects that communicate intensely may be located on the same node. By controlling the placement of objects, the programmer can exploit the concurrency between computations on different nodes.

Objects in Emerald communicate via invocations. Arguments to an invocation can be references to other objects. Passing local references in a remote invocation can possibly generate further remote invocations. To avoid performance degradation, systems like Argus [Herlihy, Liskov 82] stipulate that arguments to remote calls may only be passed by value. ESP allows passing objects by reference but disregards the performance issue. In Emerald, objects are mobile and it is possible to move an argument object to the site of the callee. Movement of an object with an invocation can be effected explicitly by the programmer (call-by-move). Based on the type information available, the compiler may decide to generate code to move the object upon invocation. Since there is a very tight binding between the compiler and the run-time system, the compiler has sufficient information to make that determination.

3.5.2. Avalon/C++

Avalon/C++ [Detlefs et. al. 88] is an experiment in building reliable distributed systems by using the inheritance properties of a language. Avalon/C++, developed at Carnegie Mellon University, uses inheritance mechanisms to customize the synchronization and fault-tolerant properties of new objects. These properties are provided by the base class hierarchy of the language (Figure 3.2).

The recoverable class provides a means of creating persistent, and hence recoverable, types. The atomic and subatomic classes provide different flavors of atomicity. Objects and classes derived from these base classes are guaranteed atomicity, i.e., their transactions execute in a serial order. Computation in Avalon/C++ is accomplished via transactions. For the system to be reliable, transactions have to be atomic, consistent and persistent. The designers of Avalon/C++ argue that the above properties cannot be

inherited automatically [Detlefs et. al. 88]. The base classes provide methods which implementors can use to guarantee these properties. For example, the implementor of an atomic_set class would derive from, say, the atomic class. To implement the operations of the atomic_set class, he would explicitly use the locking primitives inherited from the atomic class. Users of the atomic_set class will be guaranteed atomicity without having to provide explicit synchronization.

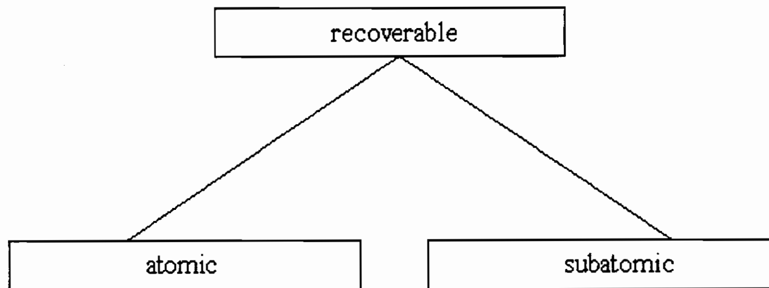


Fig 3.2 Inheritance Hierarchy of Avalon/C++ Base Classes

An application in Avalon/C++ consists of a set of servers. Each server is a collection of objects derived from the class hierarchy described in Fig. 3.2. Servers communicate by operation calls. They are implemented as remote procedure calls, where arguments and results are passed by value.

3.5.3. Choices

Choices (Class Hierarchical Open Interface for Custom Systems) [Campbell et. al. 87] is an operating system family built for the EOS project at the University of Illinois, Urbana-Champaign. Choices attempts to provide a customized operating system for a given hardware platform and application. In a computing system composed of different applications, each application runs on its own custom Choices operating system. The customization is achieved by selecting portions of the class hierarchy from which Choices is built. The designers of Choices have used C++ to built a class hierarchy which provides them with parallelization and synchronization features. For the same reason, it is an interesting case study of using class hierarchies to implement parallelism, as opposed to modifying language features to accomplish the same. Similar principles are used in ESP, the only exception being the modification of the GNU C++ compiler to provide the overloaded →() operator with certain information necessary for message passing (refer to Chapter 4).

Tasks are the elements of computation in Choices. Each task is composed of Threads, which are lightweight processes. Switching between threads is inexpensive, since it does not involve virtual memory task switching or memory cache flushing. Threads communicate via the shared memory space. In addition, message passing schemes are also provided by the operating system for communication. Choices supports the notion of persistent objects (absent in ESP), which stay resident in memory beyond the execution of a task. Tasks may communicate through the persistent objects.

Choices has been strongly influenced by the design of several other operating systems, viz., UNIX, MULTICS, CEDAR, Clouds, etc. It is unique in offering an open interface for building custom interfaces.

CHAPTER 4

EXPERIMENTAL SYSTEMS PLATFORM

4.1. Introduction

The ESP kernel [Leddy, Smith 89] was developed at MCC (Microelectronics and Computer Technology Corporation) between 1987 and 1990 with partial funding from the Defense Advanced Research Projects Agency (DARPA) and the U.S. Department of Defense. [Smith 90] describes the purpose of the ESP project to "integrate and develop technologies that would enable the rapid prototyping and implementation of scalable, high-performance computer architectures". The focus of the work was to develop hardware building blocks that could be easily integrated to form a computer architecture. Software modules assembled in a similar manner would provide an operating system for the architecture.

According to [Surma 90], system components must have the following attributes to allow for varying hardware configuration, run-time system and application software.

- easily modifiable;
- minimal dependencies ;
- easily reconfigurable;

[Surma 90] claims that ESP has achieved its goals of generality and flexibility through its modular approach and appropriate levels of functionality. The functionality required in a given experimental configuration is dependent on the goal of the experiment. ESP has the

ability to provide only those functions that are essential to the problem. In addition, the modular approach enables easy modification of existing functions and addition of new functionality to the system.

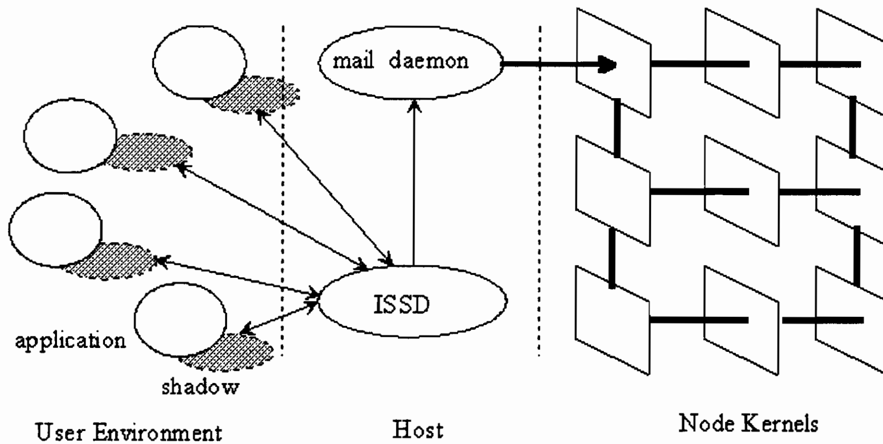


Figure 4.1 The ESP Programming Environment

4.2. Run-time Environment

The ESP run-time environment is composed of the following components:

1. shadow processes;
2. the interactive shell support daemon (ISSD);
3. the mail daemon;
4. node kernels;

The connectivity between these components is illustrated in Figure 4.1 and described in more detail below.

4.2.1. Shadow process

The shadow process runs under the user's front-end and establishes a connection to the ISSD process to start the application. During the execution of the application, it shadows

the state of the application to the user at the front-end system. It has the ability to interpret local UNIX signals, redirect output, etc. and can, therefore, provide I/O services between the user and the application. In addition, the shadow process also provides facilities for monitoring and debugging applications.

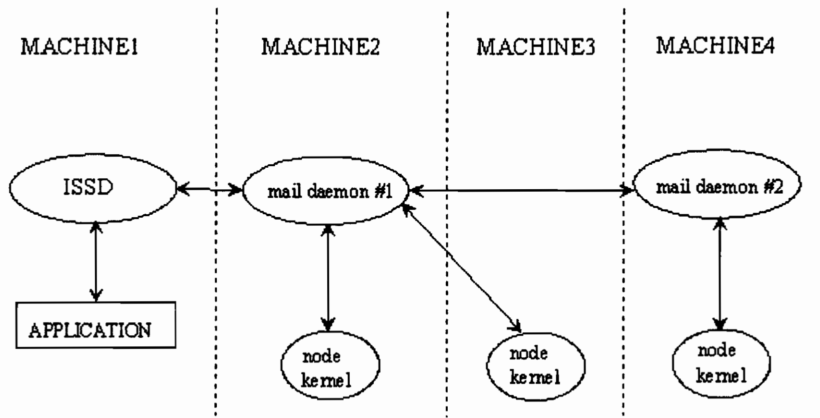


Figure 4.2 Mail Daemons and Node Kernels Connected to ISSD

4.2.2. ISSD

The ISSD process establishes a connection between the front-end processes and the mail-daemon process. [Surma 90] describes the role of the ISSD as a gateway between the front-end system and the ESP experimental system. Services offered by the ESP kernel and its PSOs (Public Service Objects) are invoked by the ISSD. The ISSD process supports several commands to control and monitor the system. These include commands to check process status, identify users logged onto the system and monitor message traffic.

4.2.3. Mail Daemon

The mail daemon runs on the host system and passes messages between the ISSD and other mail daemons or node kernels [Leddy et. al. 88]. As shown in Figure 4.2, the mail daemon connects the ISSD with other mail daemons or other node kernels. The mail daemon is responsible for mail delivery between different node kernels.

4.2.4. Node Kernels

The kernel process runs on each node of the ESP system and provides the basic services required by the application. The kernel performs message management, memory management, method execution and synchronization. All other services are provided through dynamically linked public service objects (PSOs).

4.2.4.1. Memory Management

Memory management by the kernel supports the needs of the kernel, its PSOs (public service objects) and applications. Applications request space for structures on the stack as well as on the heap. Stack-resident structures are allocated space on the method execution stack. For dynamic memory allocation, the kernel provides its own alternate definitions for malloc and free to control memory allocation and deallocation requests. The ESP system uses two memory management schemes : a modified buddy algorithm and a contiguous allocation scheme. Whenever an application creates or destroys objects, memory is automatically allocated and deallocated for the objects.

4.2.4.2. Dynamic linking and loading

In a dynamic environment like ESP, loading all the classes would be wasteful of memory. For this reason, the ESP kernel supports dynamic loading of classes. Whenever

an application requests the instantiation of an object by using the new {} operator, the kernel determines if an instance for that class exists at the node for the current application. If there is no instance of the given class, the object code for the class is loaded and linked. References to kernel symbols are resolved at this time.

4.2.4.3. Message management:

An object in ESP communicates with another object by invoking the other object's method. The method invocation is accomplished by sending a **request** message to the object whose method is invoked. When the invocation is complete, the results are returned by the callee through a **response** message. An application in ESP can request remote instantiation of an object via a **construct** message. All messages are trapped by the ESP kernel in its message loop. Depending upon the type of the message, the kernel performs appropriate processing.

When the kernel processes a request message, it uses the handle (unique identifier) of the object to obtain a local pointer to the instance of the object. It uses the method identifier in the message to locate the method code for that object. If the object is currently not executing a method, the kernel pushes the method arguments onto the stack and calls the local method. When the call is complete, the kernel checks the future structure (discussed in next section) associated with the request message to see if a reply is necessary. In that case, the kernel creates a response message and mails it to the kernel of the caller. When the kernel on the caller node processes this response message, it checks the status of the future structure on the local node. The future structure can be in one of the following three states:

- Not Blocked: The requesting object does not require a result yet. The kernel simply copies the result into the future (discussed in the next section) structure and marks it as responded.
- Waiting: The requesting object is blocked waiting for a reply. The kernel copies the result into the future structure and restores the state of the suspended instance.
- Responded: The requesting instance has received the value. No action is necessary.

On receipt of a construct message, the kernel uses the handle and the method identifier of the object to invoke the constructor for the method. The constructor for every object performs a return using a macro CONS_RETURN. The macro returns the handle of the object created to the kernel. The kernel formats a response message with the handle of the newly created object and sends the message to the reply destination.

4.2.4.4. Task Management:

The ESP kernel performs multitasking. When a method of an object becomes blocked waiting on a reply, it relinquishes control to the kernel. The kernel, then, performs the task switch to another method which is ready to run. The task switching is achieved by methods provided in the local_base class, viz., **do_it**(), **savest**() and **restore**(). When the kernel wishes to invoke an object's method as the result of a request message, it calls **do_it**(). The **do_it**() method saves the kernel stack pointer and frame pointer in variables. All the parameters are copied to the stack, one word at a time. Control is passed to the method instance by jumping to the method address. This address is determined by using the method identifier as an index into the virtual table for the object. The virtual table for an object is a table of method identifiers and their respective method addresses. This table is generated by the compiler. The method **savest**() is used to save the context of an object

when it gets blocked. This permits the kernel or another object method to execute. The object becoming blocked calls **savest**() passing its stack pointer as an argument. **savest**() saves the program counter, frame pointer and registers on the stack. The stack pointer is saved at a specific location. Control is returned to the kernel. The function **restore**() restores the context saved by **savest**(). The instance stack pointer and the return value (optional) are passed to the function. The kernel stack and frame pointers are saved and the instance stack and frame pointers are restored. A return is performed to the code location saved by **savest**() and instance values are restored from the stack.

4.3. Class Hierarchy in ESP

The previous section discussed the services offered by the kernel. The class hierarchy of the kernel will be described to illustrate how these services have been implemented. Compiler modifications and operator overloading has been used to provide features like object location and the ESP call/return mechanism [Chatterjee et. al. 89]. This section will demonstrate how these modifications have been used to implement ESP features. The basic class hierarchy in ESP is shown in Fig 4.3. The services provided by each class are shown in Fig 4.4. All application-level objects are derived from the remote_base class.

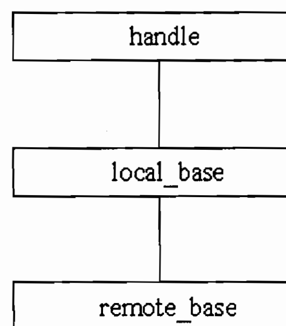


Figure 4.3 Class Hierarchy in the ESP Kernel

A handle is a unique global identifier for any object. The handle structure has been described in detail in Chapter 4, Section 4.2.3.1. Figure 4.5 illustrates the deferencing of a handle in ESP. The ESP kernel uses the handle structure to obtain a pointer to an instance of a class.

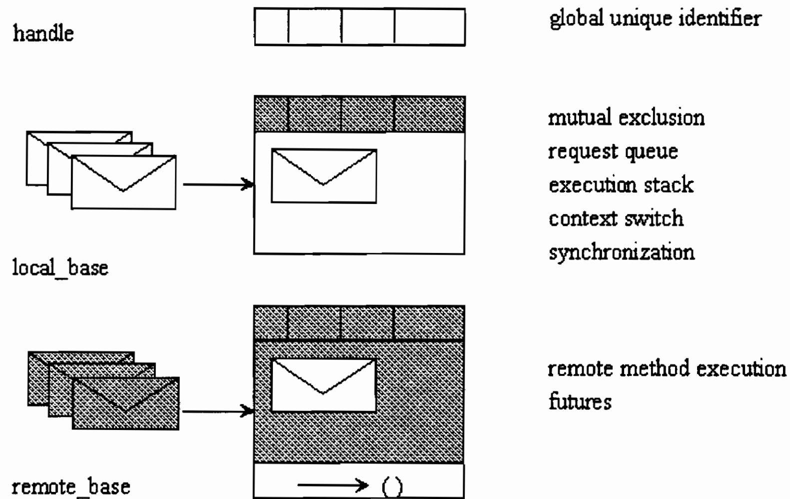


Figure 4.4 Services Provided by Base Classes in the ESP Kernel

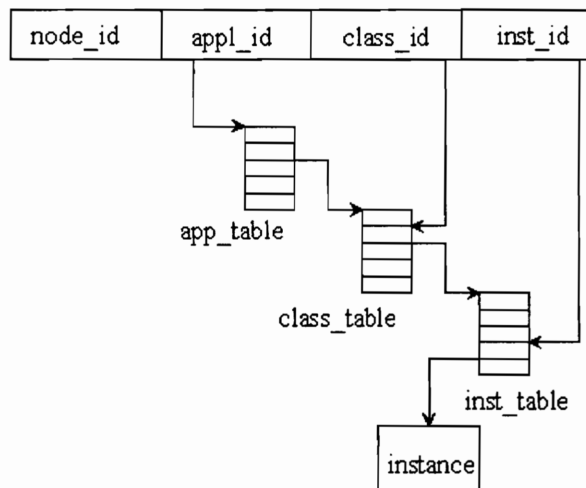


Figure 4.5 Dereferencing a Handle in ESP

The local_base class is derived from the handle class. This class provides mutual exclusion, context switching and synchronization for objects derived from it. The locking mechanism offered by local_base ensures that, at any given time, only one method of an object instance is being executed. Requests for an active object are queued by the request queue.

The remote_base class contains the definition of the overloaded →() (pointer-to-member) operator. Method invocation semantics of ESP are implemented via this overloading. The designers of ESP have modified the G++ compiler to provide them with additional information in the definition of the overloaded →() operator. If the method foo() of an object of class OBJ1 is invoked, the invocation looks like:

((OBJ1 *)objptr)→foo (arg1, ..., argn);

Within the definition of the overloaded →() operator, the compiler supplies the following information:

- the object pointer
- the virtual table index of the function
- length of the argument list
- pointer to the beginning of the argument list

In standard C++, when such an overloading is defined for a class, the compiler performs the usual type checking and type conversion operations. This implies that for every method of every class derived from remote_base, a unique definition for the overloaded →() operator would have to be supplied. This places an unreasonable burden on the application designer. To avoid this restriction, the C++ ellipses form ("....") is used. After the first two arguments (the method id and the argument length), the ellipses match all

possible argument lists. The information provided by the compiler is used by ESP to format a request message and mail it to the node of the object whose method is being invoked.

ESP provides the programmer with the facility of specifying the location of an object at the time of its creation. This is accomplished by overloading the object creation operator new for the remote_base class to allow specification of object location. The format of the overloaded new operator looks like:

objA *A_inst = (objA *) new {*location specifier*} (arg1, arg2, ..., argn);

The location specifier may be relative to the present location (*local* or *remote*) or relative to some other object or node (*sameas* or *different*). The default location is *anywhere*, in which case, the kernel uses a resource management algorithm to decide the location of the object.

4.4. Futures

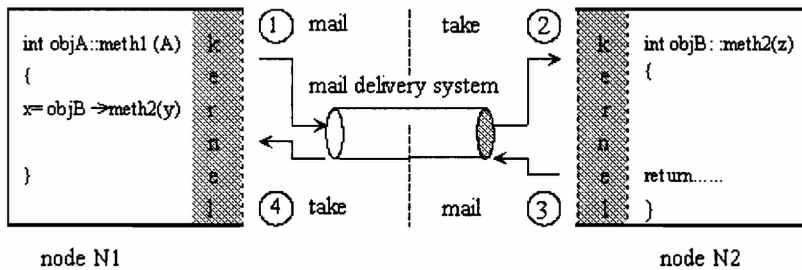


Figure 4.6 Remote Method Invocation

Before describing the future mechanism in ESP, it is necessary to understand how the kernel implements the remote method invocation in ESP. Figure 4.6 describes all the

steps in the invocation [Chatterjee et. al. 89]. The invocation depicted in the figure is a blocking invocation, i.e., the instance objA makes an invocation of the method meth2 of objB and blocks immediately. The pointer-to-member (→()) function is overloaded in the base class of objA. This function constructs a request message which contains the arguments passed to the method, the virtual table index of the method, the handle of the target object and the return information for constructing the response message. The request message is mailed and the overloaded →() operator method exits. At the target node (N2), the receiving kernel unpackages the message. It dereferences the handle passed in the request message to get a local pointer to the local instance instb. The kernel checks if the instance is currently executing any of its methods. If the instance is busy, the message is enqueued on its list of request messages. The kernel guarantees mutual exclusion for objects and ensures that the messages are processed in the order of their arrival. If the instance is not busy, then the kernel allocates a stack for the instance and pushes the arguments on the stack. The virtual index of the method is used to find a pointer for the method code and control is switched to the method.

When the message invocation is complete, control is returned to the kernel at node N2. If a reply is necessary, the kernel packages the return values into a response message and mails them to node N1. When the kernel at node N1 receives the response message, it checks the status of objA. If objA is blocked waiting for the reply, (as in Figure 4.6), the kernel reactivates meth1 of objA and passes it the results returned in the response message.

Futures play a significant role in the remote method invocation described above. In ESP, concurrency and synchronization is achieved by futures [Chatterjee 89]. When an object

in ESP makes a remote method call, it is intercepted by the kernel. A new thread of execution is created at the remote site for the object whose method is invoked and a future is returned to the caller object. The future is a reference to the results of the remote method call which will be available some time in the future. Concurrency is achieved since the caller object does not block until it actually attempts to evaluate the future. At that time, if the results of the computation are not available, the kernel suspends execution of the caller object and saves its state. When the kernel receives the results for the computation, it resumes execution of the blocked object. If the results have already been received by the kernel when the caller object attempts to evaluate the future, the results are returned to the object and the object does not block. The future mechanism allows concurrent execution of the caller and callee objects, until the caller object explicitly requests the results.

Let us suppose that an object objA wishes to invoke method foo() of object objB. In ESP, an object may perform a remote method invocation in one of the following ways:

1. ((objB*) inst_objB)→foo (arg1, arg2, ..., argn);

In this case, objA simply wishes to invoke method foo() of objB and does not care about the results of the computation.

2. future fu = ((objB *)inst_objB)→foo (arg1, arg2, ..., argn);

```
    // do some work
int Result = fu;
```

When the method is invoked, the kernel invokes the remote method foo() of objB and returns the future fu to object objA. In this case objA is not blocked as a result of the invocation. It can continue to do other work, until it attempts to evaluate the future. The kernel blocks objA if the results are not available at that time.

3. int Result = ((objB *)inst_objB)→foo (arg1, arg2, ..., argn);

In the above invocation, objA performs a blocking invocation. It invokes method foo() of objB and immediately requests the return result. The kernel blocks objA until the results from foo() become available. In the above invocation, as in the invocation of example 2, a future is created by the kernel. However in this example, its role is implicit.

CHAPTER 5

THE ACTOR MODEL OF CONCURRENCY

5.1. Introduction

Chapter 2 discussed several models of concurrency and surveyed several concurrent languages based on the actor model of concurrency. In this chapter, the actor model of concurrency [Agha 86a] will be discussed in detail. The actor model is of special interest in this project, since the focus of this work is to integrate the actor model into a distributed object-oriented environment. In the context of such systems, the primitive actor model has to be modified for reasons of performance and practicality [Kafura 88]. Modifications to the primitive actor model are discussed in detail. These modifications are relevant, since the design of ASP is based on the modified actor model as proposed by [Kafura, Lee 89b].

5.2. The Primitive Actor Model

5.2.1. Elements of the actor model

The primitive actor model is composed of five basic elements:

- actors
- mail queues
- messages
- behaviors
- acquaintances

[Kafura 88] defines an actor as a concurrent, autonomous computing agent interacting with other actors exclusively through the exchange of messages. An actor is uniquely identified by its mail queue address, which is the only information necessary to send the actor a message. Messages sent to the actor's mail queue are processed in strict FIFO order. The mail queue address of other actors may be passed as part of the message. This allows an actor system to be dynamically reconfigurable [Agha 86b] (discussed in Chapter 2).

A behavior is defined by a set of acquaintances and a script. The acquaintances of a behavior are the actors whose mail queue addresses are known to the behavior. Acquaintances consist of :

- actors to whom results may be returned; or
- actors who may provide certain services; or
- actors that model the state of the behavior.

In object-oriented languages, acquaintances correspond to instance variables. However, in Agha's model, acquaintances are read-only variables. This requirement is necessary since the actor model assumes inherent concurrency, i.e., every statement of a behavior script can be executed in parallel. To achieve inherent concurrency, the actor model provides only side-effect free operations. The script of a behavior is a set of instructions which describe how a message will be processed. The script corresponds to a class definition in object-oriented languages.

Dynamic reconfigurability is achieved by the use of acquaintances. An actor can know about acquaintances either statically or dynamically. In the static case, a list of

acquaintances is bound to the actor at compile-time. In the dynamic case, there is a two-step process by which an actor can be bound to an unknown acquaintance. First, the actor receives the address of an actor as part of a message. Next it includes this address in the set of acquaintances specified in the become operation. Thus it is possible for the actor to acquire an acquaintance dynamically.

5.2.2. Message processing in actors

When an actor is created, an initial behavior is created and bound to the actor. The first message on the actor's mail queue is removed and bound to the initial behavior. The behavior executes its script, and at some point in the execution specifies a replacement behavior, by means of the become operation. The replacement behavior is now the current behavior of the actor and is ready to process the next message in the queue. If there is a message waiting for the behavior, the message is bound to the new behavior, otherwise, the behavior awaits the arrival of a message. If a message arrives when there is no waiting behavior, it is queued on the actor's mail queue. Each behavior processes exactly one message and terminates. Each message is bound to exactly one behavior. This cycle continues until the actor terminates itself by specifying a null behavior or the actor is garbage collected.

When processing a message, an actor can take the following actions:

- create a new actor;
- send a message to another actor; and
- specify a replacement behavior.

When an actor creates a new actor, a mail queue address is returned. The actor can use this mail queue address to send messages to the newly created actor. The mail queue

address can also be passed in messages to other actors, thus making the existence of the new actor known to other actors that may then send messages to the new actor.

The send primitive is used to send messages asynchronously. The message contains the mail queue address of the actor, the name of the method to be invoked and the parameters for the method invocation. The sender actor is not blocked by the message send, since the mail queue of the receiving actor buffers messages.

The actor can use the become operation to specify a replacement behavior (Figure 5.1). The operation requires a behavior script name and a list of acquaintances. A new behavior is created and bound to the actor. During its lifetime, an actor may be bound to several different behaviors. The behavior currently bound to the actor is known as its current behavior. A behavior may perform the become operation only once in its lifetime. Hence, at any given time, the actor has only one current behavior. This behavior determines how the next message on the actor's mail queue will be processed.

The become operation plays an important role in the actor model. It is the mechanism by which an actor can change its state. Since acquaintances cannot be written to, the actor can only change state by doing a become and specifying a new set of acquaintances. Additionally, the become operation also works as a synchronization mechanism. The specification of a replacement behavior by the become operation allows the next message in the actor's mail queue to execute the behavior script. When a behavior is executing in its script, all messages on the actor's mail queue are queued until the behavior does a become. In addition, inter-object concurrency can be achieved by the use of the become

operation. Inter-object concurrency arises when the behavior does a become before the end of its script. This allows multiple behaviors of an actor to execute concurrently.

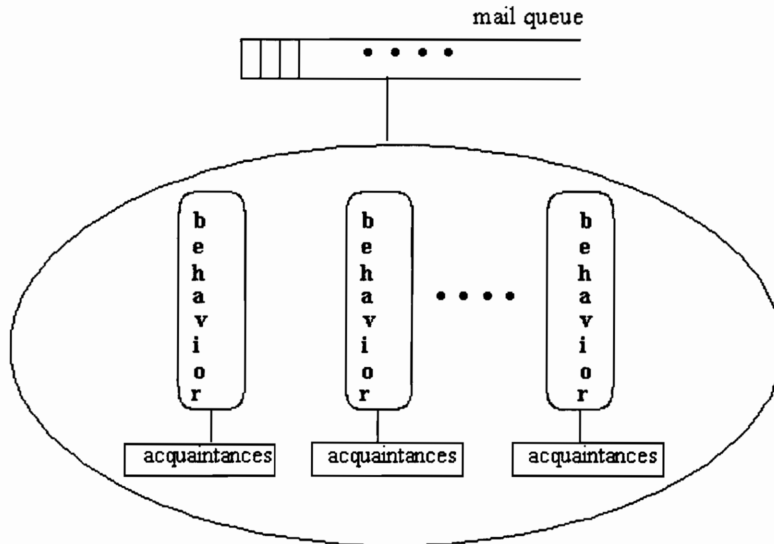


Figure 5.1 An Actor

Depending on the placement of the become operation, actors are classified as serialized and unserialized actors. A serialized actor is one that enforces strict ordering in its processing of messages. In such an actor, the become operation is placed at the end of the behavior script. On the other hand, an unserialized actor can, at least partially, have many behaviors active at the same time. Such an actor would exploit the potential for inter-object concurrency by placing the become operation in the middle or at the beginning of its script.

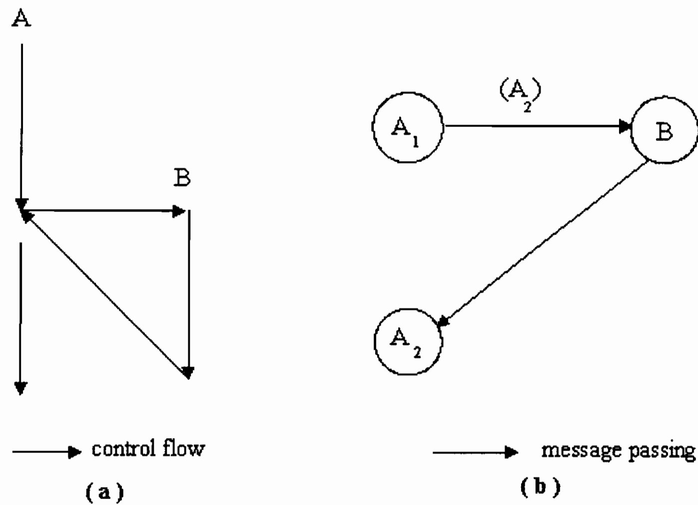


Figure 5.2 Continuations

5.2.3. Continuations and customers

Figure 5.2 illustrates the concept of continuations [Kafura 88]. Shown in Figure 5.2(a) is a typical call/return relationship between procedures A and B and the same structure is depicted in Figure 5.2(b) with the use of actors. Actor A1 sends the mail queue address of actor A2 in a message to B. The intention is to forward the results of the computation by B to A2. Actor A2 is called the continuation because it is the logical extension of the work started by A1. The call/return structure in Figure 5.2(a) is identical to the one in Figure 5.2(b) because:

- the mail address of A2 passed to B is analogous to the return address of the procedure A; and
- the continuation A2 is analogous to the statements following the return, since those statements are the logical extension of the work preceding the procedure call

The similarity ends here; the call/return structure of functions A and B is fixed. It can be modified in case (b) if actor A1 sends the address of a different continuation to B at a

different time. Also, the continuation may not be used. For example, in the case when the continuation is used to handle only exceptional conditions.

The notion of a customer is also based upon the passing of mail queue addresses in messages. Consider the often used boss-worker example. The client requests a certain service from the boss actor, which distributes the work among the worker actors. The boss actor passes the address of the client actor in its message to the worker actors. After the computation is done, the worker actors can reply directly to the client actor. The client actor is the customer, since the computation is being performed for it and it is the recipient of the final result.

5.3. Modifications to the Primitive Actor Model

As discussed in the introduction, performance considerations necessitate changes to the primitive actor model. This section will discuss these modifications in detail. The modified actor model is called the ACT++ model.

5.3.1. Granularity of actors

The granularity of actors in the primitive actor model [Agha 86a] is of the level of an instruction. This is suitable for an architecture with a large number of processors connected by high-speed links. The instruction-level concurrency is not very useful in application domains where the target architecture is a small number of loosely coupled processors. For this reason, a decision was made [Kafura, Lee 89b] to adapt the actor model to use a more coarse-grained structure. Every actor was a light-weight process, i.e., the size and life-time of a behavior was of the order of a procedure.

Actors in the ACT++ model are no longer subject to the constraints described in Section 2.1. The programming paradigm is similar to an imperative programming language, as opposed to the functional paradigm stipulated by Agha's model. The actor methods are no longer required to use side-effect free operations. Acquaintances can be written to; this allows the actor change its state without specifying a replacement behavior.

Another important distinction, is that the ACT++ model distinguishes between passive and active objects (like ABCL/1 [Yonezawa et. al. 87]). Data types like integers are modeled as passive objects. The distinguishing factor between active and passive objects is that active objects have their own thread of control. Passive objects have no thread of control and can only be invoked by other active objects. An instance of an active objects is an actor, which can execute independently of other objects.

5.3.2. Reply messages and Cboxes

The ACT++ model uses Cboxes as a mechanism for receiving the result of a method invocation. The Cbox, named after the Cbox structure of Concurrent Smalltalk [Yokote, Tokoro 86], is a special kind of mailbox used to retrieve replies from a method invocation. The sender of the request message (client) includes the Cbox name in the message. The actor performing the method invocation (server) can reply to the Cbox or forward it, i.e., pass the Cbox name to another actor. When the requesting actor attempts to read from the Cbox, if a reply is available in the Cbox it is delivered to the actor, otherwise the caller is blocked until a reply arrives.

Cboxes have been incorporated into the ACT++ model. Cboxes obviate the need for creating continuations to await the result of an invocation. They provide a channel for the

sender of a request message to receive the reply from an invocation, without having to create continuations to wait on the reply.

Cboxes provide a consistent interface to the server actor to return replies. Using continuations, the server actor needs to have prior knowledge about the client actor. It needs to know what method of the client actor to invoke in order to send it a reply. Using Cboxes, the server actor can make the assumption that every client actor provides the name of a Cbox for returning replies. In Agha's model, a client waiting for multiple results has to create separate continuations in order to differentiate between the different results. It is not necessary to create separate methods to process different results when using Cboxes. A client can associate a different Cbox with every request message and achieve the desired semantics.

Cboxes provide a efficient way of modeling actors which cannot proceed until they receive a reply. In the primitive model, this is modeled by means of an *insensitive* actor. An actor becomes insensitive by assuming a behavior that forwards all messages to a buffer actor. When the actor receives the reply message it is waiting for, it sends a message to the buffer actor to send back all the queued messages. Cboxes are a more flexible mechanism than insensitive actors because an actor may create many Cboxes before waiting on any one of them. In addition, multiple request messages may be associated with a single Cbox. In this case, an actor can choose to wait on just one reply (or-synchronization) or wait on all replies (and-synchronization).

Cboxes can also be used to implement the semantics of customer actors. To create a customer actor, a client actor may pass a Cbox to the constructor of the customer actor.

The client actor can then specify the same Cbox as a reply destination of a request message sent to a server actor. This will result in the replies from the server actor being sent to the customer actor.

CHAPTER 6

DESIGN OF THE ACTOR SYSTEMS PLATFORM

6.1. Introduction

The Actor Systems Platform (ASP) is designed to provide an actor-based concurrent programming environment over ESP, a distributed object-oriented system. ASP is a platform for developing applications based on the modified actor model proposed by [Kafura, Lee 89b]. This model is discussed in the previous chapter. The present chapter discusses the design of ASP in detail. The design of the class hierarchy to achieve the actor abstraction is presented. The use of the pointer-to-member-function (→()) operator, which is critical to the design of ASP, is described. The implementation of Cboxes, which provide a synchronization mechanism for receiving replies, is discussed. The implementation of behavior sets in ASP is described. Finally, some of the features of ASP are illustrated by examples.

6.2. ASP : an application of ESP

An application in ESP consists of a set of classes derived from the base class, **remote_base**. We have designed ASP as a hierarchy of classes derived from the **remote_base** class (Figure 6.1). The actor abstraction is provided by designing an application layer above ESP. The kernel or other components in ESP are not modified to implement the actor abstraction. This enables the system to be easily installable and portable across different versions of ESP. Message passing semantics of the actor model are provided by redefining the the pointer-to-member-function operator (→()) in the

behavior class. All objects derived from the behavior class in ASP will inherit the actor semantics of message passing. It is possible to run non-actor ESP applications on the same platform. These applications would be composed of objects derived from the ESP base class. Objects in ASP may use the explicit return mechanism of Cboxes or the implicit return mechanism of futures provided by ESP to wait on replies. The ASP class hierarchy provides an additional layer of functionality on ESP, without taking away the ability to run ESP applications.

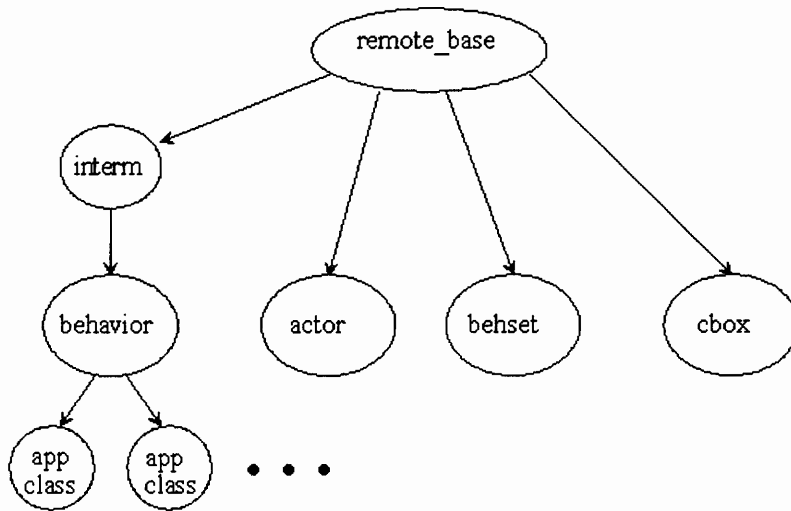


Figure 6.1 Class Hierarchy in ASP

6.3. Class Hierarchy in ASP

The class hierarchy in ASP is shown in Figure 6.1. The **actor** and **behavior** classes are discussed in this section. The class declarations for these classes are shown and discussed. Each action performed by an actor is described along with the code associated with the action. The lifecycle of the actor system is discussed and the necessary terms are defined in the previous chapter. This chapter provides a detailed explanation of all actions performed in the actor system.

6.3.1. Class declarations

This section discusses the class declarations for the **actor** and **behavior** classes. A brief description of these classes is provided to set the context for the discussion in the subsequent sections.

The actor class provides the mail queue abstraction in the actor model. All requests for method invocation are buffered by the actor. The behavior of an actor defines the script of the actor. A script is a set of instructions describing how a request is processed. It is important to note the distinction between the actor and behavior class; the actor simply buffers all messages, whereas the behavior provides the definitions of all methods which define the script of the actor.

The class declaration for the actor class is shown in Example 6.1. The actor class maintains a pointer (**mail_q_ptr**) to its mail queue where all the request messages are buffered. Methods are provided to add or delete messages from the mail queue (**add_actor_req**, **get_actor_req**). The actor class stores a handle to its most current behavior (**most_curr_behavior**) and a boolean variable indicating if it has a waiting behavior (**waiting_behavior**). The behavior set computed for the most current behavior is also stored. The method **receive_req_msg** is invoked when a new message arrives on the actor's mail queue. The method **deliver_req_msg** is invoked when a behavior specifies a replacement behavior. These methods are discussed in detail in the next section.

```
          EXAMPLE 6.1

class actor:public remote_base
{
   actor_req*  mail_q_ptr;       /* pointer to the actor's mail queue
   */
   bool   waiting_behavior;  /* is there a waiting behavior       */
   handle most_curr_behavior;/* handle to waiting behavior        */
   handle    curr_behset;        /* behavior set of most curr beh       */
protected:
   local_streams inout;
public:
   actor(handle);   /* constructor for class actor       */
   ~actor();      /* destructor for class actor       */
   __future_ref__ receive_req_msg (int, int, ...);
   int deliver_req_msg (handle);
   void add_actor_req (request_msg *, int);
   request_msg *get_actor_req();
   int update_curr_beh(handle);
}

//******************************************************************
 // Constructor for the actor class:-
 //    This user invokes the constructor of a behavior by New(beh)
 //    which gets expanded to new actor(new beh); The new operation
 //    on the behavior returns a handle to the behavior and the
 //    actor's constructor gets invoked with the behavior handle
 //    passed to it as an argument

//******************************************************************
actor::actor(handle init_beh) : inout(this)
{
   interm* interm_inst;

   mail_q_ptr = NULL;
   waiting_behavior = TRUE;
   most_curr_behavior = init_beh;

   interm_inst = (interm *) &init_beh;
   curr_behset = interm_inst->set_parent(*(handle *)this);

   CONS_RETURN;
}

//******************************************************************
//Destructor for the actor class
//******************************************************************
actor::~actor() { }
```

```
//*****************************************************************
// Receive_req_msg:
//   This method is invoked by the receiver's node kernel. If
//   a waiting behavior exists for the actor, then the msg is
//   sent to the behavior, else it is queued up on the
//   actor's queue.
//*****************************************************************

__future_ref__ actor::receive_req_msg(int meth_id, int arg_len, ...)
{
  behset *curr_beh_ptr;

  if (waiting_behavior)
  {
    /*
      then send the request to it
    */
    request_msg* pmsg  = new request_msg (&most_curr_behavior,meth_id,

  arg_len,(int*)&arg_len+1,NULL);

    /*
      Check if the method is locked or not before mailing it
    */
    if (((behset *)&curr_behset)->is_locked(meth_id+METH_OFFSET))
      add_actor_req(pmsg, meth_id);
    else
    {
      waiting_behavior = FALSE;
      POST_OFFICE->mail(pmsg);
    }
  }
  else
  {
    /*
      create a request_msg and queue it up on the actor's mail queue
    */
    request_msg* pmsg = new request_msg (&most_curr_behavior, meth_id,
               arg_len, (int*)&arg_len+1,NULL);
        add_actor_req(pmsg, meth_id);
    pmsg = NULL;
  }
}    /* end of function receive_req_msg  */


//*****************************************************************
// Deliver_req_msg:
//   This method is invoked by a newly created behavior (as a result
//   of the become operation). The behavior is passed a request_msg
```

```
//   if one exists and the actor's most_curr_beh info is updated
//   OR the actor simply records the fact that there is a
//   most_curr_beh and its handle is stored in waiting_behavior
//*************************************************************************

int actor::deliver_req_msg(handle beh_ptr)
{
   request_msg* pmsg;

   update_curr_beh(beh_ptr);

   /*
      Go through the message queue and try to find a viable message,
i.e., one
      that is not for a locked method
      If there is such a message on the actor's queue, send it to the
newly
      created behavior
   */
   if (pmsg = get_actor_req())
   {
      waiting_behavior = FALSE;

      pmsg->node_id = most_curr_behavior.node_id;
      pmsg->appl_id = most_curr_behavior.appl_id;
      pmsg->clas_id = most_curr_behavior.clas_id;
      pmsg->inst_id = most_curr_behavior.inst_id;
      POST_OFFICE->mail(pmsg);
   }
   else
      waiting_behavior = TRUE;
}
```

The class declaration for the behavior class is shown in Example 6.2. All application objects in ASP derive from the behavior class (Figure 6.1). The behavior class provides the definition of the overloaded pointer-to-member function. This overloading provides the actor message passing semantics in ASP. Any class that derives from the behavior class inherits the special overloading. Application objects in ASP can specify a replacement behavior using the **become** operation provided in the behavior class.

As shown in Figure 6.1, the behavior class derives from the interm class. The redefinition of the pointer-to-member-function operator (→()) in the behavior class causes all messages to a behavior to be routed to its parent actor. However, in the actor system, the actor and its behaviors need to be able to communicate directly, as in the case of the **set_parent** method. This method is used to inform the behavior about its parent actor. In this instance, the actor needs to directly invoke the **set_parent** method of its behavior. The interm class is defined in order to bypass the special operator overloading defined in the behavior class.

The methods of the actor and behavior classes are described in greater detail in the following sections.

6.3.2. Actions performed in an actor system

6.3.2.1. Creating an actor

When an application wishes to create an actor, it does so by using the special New operator provided by ASP. The syntax of the New operator is as follows:

```
actor *actor_inst = New ( beh (arg1, arg2, ...., argn) );
```

where,

actor_inst is the handle of the actor created by the New operation;

beh specifies the class of the behavior that is associated with the actor;

arg1, arg2, ...,argn are arguments to the constructor of the behavior.

The New operator is implemented as a macro in the actor class, which expands the above statement to:

```
actor *actor_inst = new {loc_id} actor (new {loc_id} beh(arg1, arg2, ...., argn) );
```

where,

loc_id is the location specified for creating the object (refer Chapter 4 Section 3).

The effect of the New operation is to create an initial behavior for the actor and pass its handle (refer Chapter 4 Section 3) to the constructor of the actor. The definition of the constructor of the **actor** class is shown in Example 6.1. The actor stores the handle of the newly created behavior as its most current behavior. It sets its **waiting_behavior** flag to TRUE; any request messages received by the actor will be routed to its most current behavior. The actor invokes method **set_parent** of its initial behavior to supply it with a handle to its parent actor. At any instant in the execution of an ASP application, every actor knows the identity of its most current behavior and vice-versa. This is necessary in order to implement message passing and behavior replacement in ASP.

6.3.2.2. Send a message to another actor

When an object in ASP (client) wishes to obtain a service from an actor object (server), it invokes a method of the behavior associated with the actor. The syntax of the method invocation is:

((beh *) inst_actorA) →foo(arg1, arg2,....argn);

where

beh is the behavior associated with actorA;

foo is the behavior method being invoked;

arg1, ..., argn are the arguments to the behavior's method.

```
        EXAMPLE 6.2

class behavior:public interm
{
  local_streams    inout;
public:
  behavior();    /* constructor for behavior class */
  ~behavior();     /* destructor for behavior class  */
  __future_ref__ operator->() (int, int, ...);
  int become(handle):
};
//**********************************************************************
//Operator->():
//   Overloading of -> operator redefined for class behavior
//**********************************************************************

__future_ref__ behavior::operator->()(int meth_id, int arg_len, ...)
{
   request_msg* pmsg;
   int new_arg_len = arg_len + sizeof(int) + sizeof(int);
   pmsg = new request_msg ((handle*)this,
     (int)&actor::receive_req_msg, new_arg_len,(int*)&meth_id,NULL);
   return ((__future_ref__)pmsg);

}         /* end of operator->() overloading    */

/**********************************************************************
****
  BECOME
**********************************************************************
***/
int behavior::become (handle new_beh)
{
  ((actor *) &get_parent())->deliver_req_msg(new_beh);

  INT_RETURN (1);
}
```

The type-casting in the above call is required since the method foo() is defined in the behavior class, beh. The pointer-to-member-function operator (→()) is overloaded in the behavior class. The overloading of the →() operator captures the above invocation and redirects the message to a special method of the actor managing the mail buffering

(**receive_req_msg**). The GNU compiler has been modified to allow overloading of the →() operator. When the →() operator is invoked, two additional arguments are pushed on top of the stack (Figure 6.2):

- vtable (virtual table) index of the method being invoked (meth_id)
- number of bytes of parameter data (arg_len)

&actor::receive_req_msg
new_arg_len
meth_id
arg_len
arg1
arg2
arg3
• • •
argn

Figure 6.2 New Request Message Constructed

As shown in the definition of the method, a request message is constructed with the above items and the arguments to the function. The name of the actor's mail buffering method **receive_req_msg** is inserted in the message. When the method **receive_req_msg** of the actor is invoked, it extracts the identifier of the method invoked and the length of the arguments passed to the method (Example 6.1). The **receive_req_msg** method constructs a request message from the extracted method identifier and argument length. If there is a waiting behavior for the actor, the message is forwarded to the behavior. If there is no waiting behavior for the actor, the actor enqueues the message in its mail queue.

6.3.3. Specify a replacement behavior

An object in ESP can specify a replacement behavior by invoking the **become** method of class behavior. The call looks like:

become (new beh (arg1, arg2 ...., argn))

where,

beh is the class of the behavior;

arg1,....argn are arguments to the constructor of the behavior class.

A new behavior object is created and its handle is passed to the method **become** of class behavior. The newly created behavior needs to inform the parent actor about its existence. This is accomplished in the behavior class by invoking the method **deliver_req_msg** of the actor (Example 6.1, Figure 6.3). The actor checks its mail queue for pending requests. If there are requests in the mail queue, it sends the first request message to the newly created behavior. If the actor's mail queue is empty, the actor simply updates its current behavior information.

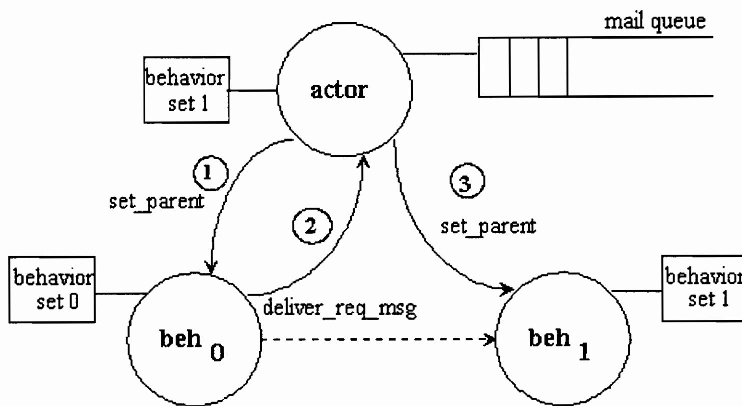


Figure 6.3 Message passing Between Actor & Behavior

6.4. Cboxes: A Mechanism for Synchronization and Returning Replies

As described in Chapter 4, ESP provides futures for synchronization and returning results. Futures provide the essential functionality of Cboxes. It is possible to block on a future waiting for a result. A future is a first-class object that can be passed to other objects. Managing multiple method invocations and replies can be accomplished by using future sets. Consider the scenario where an object sends messages to many objects. It may wish to block when:

- one of them responds;
- some of them respond; or
- all of them respond

The above semantics can be easily realized by using future sets. A future set consists of a set of futures and methods to manipulate the members of a set [Chatterjee 89]. It is possible to simulate future sets by creating an array of futures and polling the members of the array. However future sets provide a high-level mechanism for such manipulations.

Despite the availability of the futures mechanism in ESP, a design decision was made to develop the Cbox return mechanism in ASP. Cboxes were chosen as the reply return mechanism for the following reasons. Incorporating Cboxes into the actor platform would make ASP consistent with the ACT++ model (described in Chapter 5). In ASP, messages sent between behaviors are intercepted by the actor. This presents difficulties while using the ESP implicit return mechanism for returning results. A typical method invocation in ESP is shown in Figure 6.4(a). When messages sent to the behavior (objB) are routed to the actor (Figure 6.5), the ESP return mechanism can no longer be used. The kernel at the node where objB resides is not aware that the response message needs to sent to the node where objA resides, and not to the node where Actor resides. In addition, since the usual

call/return mechanism in ESP is being bypassed, objA does not remain blocked until objB completes the execution of method foo().

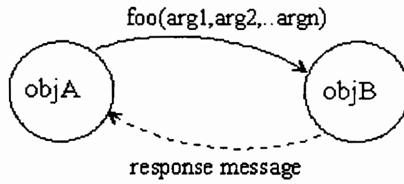


Figure 6.4 (a) Implicit Return Of Results Via Futures

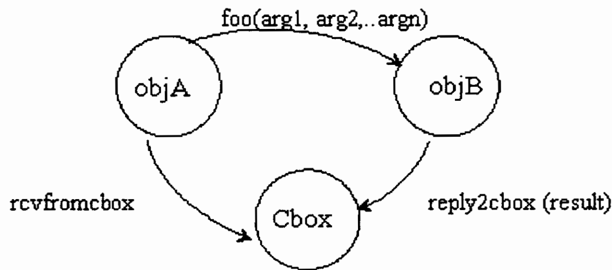


Figure 6.4 (b) Returning Result Explicitly Via Cboxes

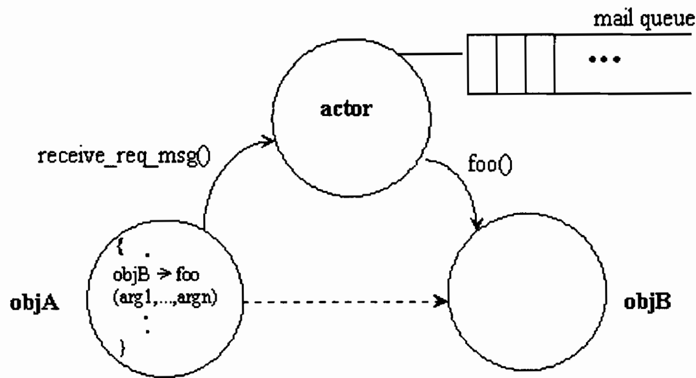


Figure 6.5 Redirection of Messages to the Actor

6.4.1. Implicit Vs Explicit Return

In ESP, when objB wishes to return results to objA, it does not explicitly invoke a method of objA. Instead, it uses a predefined macro (e.g., INT_RETURN) depending on the type of the return result. The kernel at the node of objB captures this return and constructs a response message (if necessary), and mails it to the node of the invoking object, objA. The return mechanism is implicit (Figure 6.4(a)); the structures and the mechanism used to return results are hidden from the programmer.

In the Cbox implementation, the invoking object objA creates a Cbox object and passes its handle to the invoked object, objB, along with the message (Figure 6.4(b)). To return results, objB explicitly invokes a method of the Cbox. On the receiving side, objA invokes a method of the same Cbox to receive results.

6.4.2. Design of Cboxes

The Cbox class provides a synchronization mechanism for returning and awaiting results in ASP. The declaration of the Cbox class is shown in Example 6.3. To return results, an application object invokes the method **reply2cbox** of the Cbox (Example 6.3). The Cbox defines its own overloading for the pointer-to-member-function (→()) operator, so that the length of the result being passed can be determined. The Cbox stores the results until the object which requires the results invokes the method **rcvfromcbox**. The Cbox class maintains a linked list of result structures (ResultStrPtr). Each item in the list stores the size of the result and a pointer to the result.

```
      EXAMPLE 6.3
class cbox:public remote_base
{
   local_streams inout;
   Result_T  *ResultStrPtr;
   int     ResIndex;
public:
   cbox();
   ~cbox();
   __future_ref__ operator->() (int,int,...);
   int reply2cbox(int,...);
   void *rcvfromcbox();
};
//*******************************************************************
// reply2cbox  : copies the results into its structure and
//               unlocks rcvfromcbox
//*******************************************************************

int cbox::reply2cbox(int arg_len, ...)
{
   ResIndex++;

   Result_T    *TmpPtr;

   /*
     Calculate the number of words from the number of bytes
    */
   int norm_arg_len = (arg_len + 3) >> 2;

   /*
     Fill in the result structure
    */


   if (ResultStrPtr)
   {
     TmpPtr       = ResultStrPtr;
     while (TmpPtr->Next)
        TmpPtr = TmpPtr->Next;
     TmpPtr->Next    = (Result_T *) __builtin_new ( sizeof (Result_T) );
     TmpPtr = TmpPtr->Next;
     TmpPtr->ResPtr = (void *) __builtin_new ( norm_arg_len );
     TmpPtr->ResSize= arg_len;
     TmpPtr->Next    = NULL;
     wordcopy ( &arg_len+1, TmpPtr->ResPtr, norm_arg_len );
   }
   else
   {
```

```
    ResultStrPtr      = (Result_T *) __builtin_new ( sizeof (Result_T)
);
    ResultStrPtr->ResPtr= (void *) __builtin_new ( norm_arg_len );
    ResultStrPtr->ResSize= arg_len;
    ResultStrPtr->Next = NULL;
    wordcopy ( &arg_len+1, ResultStrPtr->ResPtr, norm_arg_len );
  }

  unlock_method((int)&cbox::rcvfromcbox);

  INT_RETURN(1);
}

//*************************************************************
// rcvfromcbox : return the 1st result in the list and delete it
//               from the list
//*************************************************************
void* cbox::rcvfromcbox()
{
  Result_T    *TmpPtr;

  /*
  *  Store the 1st result structure and delete it from the list
  */

  TmpPtr = ResultStrPtr;
  if (ResultStrPtr)
    ResultStrPtr   = ResultStrPtr->Next;

  /*
   * If there are no more results in the list lock this method
  */

  if (--ResIndex < 0)
    lock_method((int)&cbox::rcvfromcbox);

  PTR_MORERET (TmpPtr->ResPtr, TmpPtr->ResSize);

}
```

The return of results using Cboxes is illustrated in Figure 6.4b. The Cbox handle is passed to the invoked object (objB) as an argument of the method foo. The object performing the computation (objB) returns results by invoking the method reply2cbox of the handle passed to it. The syntax for returning results is:

int RetCode = ((Cbox *)&inst_cbox) → reply2cbox(answer);

where inst_cbox is the Cbox handle passed to objB and answer is the result being returned. If answer is a pointer to the result, we need to be able to capture the length of the result. The pointer-to-member-function operator (→()) is overloaded in the Cbox class to obtain this information. As discussed in Section 2, in the overloading of the →() operator, the virtual table index of the method and argument length are pushed on the stack. The **reply2cbox** method extracts the length and stores it in the result structure.

When the method of objA accesses the reply from function foo, it invokes the method **rcvfromcbox** of the Cbox associated with the invocation of function foo. In **rcvfromcbox**, the result list is scanned and the first result, if available, is returned. An object may use the same Cbox for multiple method invocations and wait on that Cbox for all of the results obtained, if it does not care about the order in which the results are returned. This is demonstrated in the factorial example discussed later. When the **rcvfromcbox** method returns, the waiting object receives the result and ceases to be blocked.

ESP provides a mechanism of blocking the execution of methods of an object (method locking). When there are no results available in the Cbox, the **rcvfromcbox** method is locked. Subsequent attempts to obtain results from the Cbox will be queued, until a result becomes available. When the Cbox receives a result (by an invocation of the **reply2cbox** method), the **rcvfromcbox** method is unlocked.

One of the limitations of the return mechanism in ESP is that there is no way to return a result of arbitrary length, unless the result is formatted as a structure. The PTR_RETURN(ans_ptr) macro is provided in ESP for returning pointers. If ans_ptr

points to a structure, the macro returns the structure, otherwise the value of ans_ptr (address) is returned. ASP overcomes this limitation by providing a macro PTR_MORERET(ans_ptr, ans_len) which can return ans_len bytes pointed to by ans_ptr. The **rcvfromcbox** method uses this macro to return a result of arbitrary length. The macro sets the appropriate values in a structure, which ensures that when the ESP kernel creates the response message, it copies the entire result in the message.

6.5. Behavior Sets

6.5.1. Design of Behavior Sets

[Kafura, Lee 89a] observed a conflict while integrating concurrency in a object-oriented language with inheritance. Due to the concurrent nature of execution of the methods of an object, it is necessary to control the visibility of the object's interface. [Kafura, Lee 89a] found that a synchronization mechanism is required to block invocations of methods whose execution would violate the consistency of the state of the object. The example in the following paragraph illustrates the problem.

Consider the example of an object implementing a bounded buffer on which the operations **get** and **put** may be performed. It is not valid to execute a **put** operation when the buffer is full or a **get** operation when it is empty. Without a concurrency control mechanism, the burden would be on the developer to ensure that the state of an object was not violated. Previous approaches to concurrency control have employed either centralized or decentralized control. [Kafura, Lee 89a] discuss the above methods and argue that the problem with these methods is that it is difficult to combine inheritance with these mechanisms.

[Kafura, Lee 89a] present a solution for solving the synchronization problem using a mechanism called behavior abstractions. An object manager is associated with every active object. It is created automatically when the object is created and is destroyed when the object is destroyed. The object manager controls the interface to the object, i.e., it decides whether a request for invoking an object's method is valid or not. A method of the object is open or closed depending on whether the object can accept a message for it. A message for an open method is authorized, and a message for method that is closed is unauthorized. When an object is constructed, it specifies the set of open methods to the object manager. When the object specifies a replacement behavior, the set of open methods is recomputed.

When an authorized message arrives, the object manager invokes the method for the object and closes all other methods. While executing the method, the object may specify a replacement behavior. A new set of methods are now open for execution. The object manager needs to keep track of the changing set of open methods. To accomplish this, a new way of specifying replacement behaviors is introduced (behavior abstractions).

The object specifies its replacements behavior with reference to its behavior name. [Kafura, Lee 89a] define a behavior name as "a handle for a set of open method names". For example, the set of valid behavior names in the case of a bounded buffer actor would be:

empty buffer = {put()}
full_buffer = {get()}
partial_buffer = {put(), get()}

When a partial_buffer object becomes full, it will specify a replacement behavior by doing a "become full_buffer", thereby closing the put method. Behavior abstraction relieves the programmer from the burden of programming the concurrency control inside of each object. In addition, the concurrency can be inherited by derived classes.

[Papathomas 89a] observed that the behavior abstraction has limitations. All possible behaviors of the object have to be computed in advance. A dynamic behavior abstraction scheme would allow the object to dynamically compute its interface visibility. To address the issues discussed above, [Tomlinson, Singh 89] developed the idea of enabled sets. The development platform was Rosette, a concurrent object-oriented system running on a parallel architecture. Objects in Rosette change state by using the next command. The specification of the next state includes an enabled set, which is a "set of patterns that defines the messages that can be executed in the next state". The syntax of the next command is:

(next enabled_set [id expression].....)

where enabled_set is the set of message patterns. The term [id expression] specifies that acquaintance id would have the value expression in the next state. In Rosette, an enabled-set is a first class of object which can be passed as an argument to methods of other objects. Enabled sets provide the object in Rosette with the ability to dynamically alter its interface.

The desire to integrate the flexibility of enabled sets with the properties of behavior abstractions let to the development of behavior sets. We will discuss the design of behavior sets in the context of an actor-based concurrent language. Like enabled sets, behavior sets are first-class objects which can be dynamically manipulated. The behavior

set of an object defines the set of open methods for the object, i.e., the set of valid operations that can be performed on the object. A message for a closed method of the object will not be processed until the method is opened. The object changes its state by specifying a replacement behavior. The state of the object is specified to the new behavior by passing arguments to the new behavior's constructor. Based on its state, the new behavior can dynamically compute its behavior set. Thus, behavior sets provide the object with a way to dynamically modify its interface over the lifetime of the object.

6.5.2. Implementation of Behavior Sets

ESP provides its own synchronization facility called method locking. An object may lock or unlock methods of an instance of its class or any other instance that it knows about. This scheme suffers from the following limitations. The burden of providing synchronization is on the programmer. The synchronization is built into the object definitions and cannot be inherited. To provide the necessary synchronization in ASP, a more flexible and efficient mechanism is necessary. For this reason, behavior sets are implemented in ASP.

In ASP, behavior sets are implemented as a class derived from the ESP base hierarchy (Figure 6.1). A behavior set is created whenever a behavior is created. The behavior computes its initial behavior set based on the arguments to its constructor. The behavior set is recomputed every time the behavior does a become. By default all methods are open. The behavior can close the methods which cannot accept messages. The declaration of the behavior set class follows:

```
class behset:public remote_base
{
    local_streams     inout;
    int               guard_bits[32];
```

```
public:
    behset();
    ~behset();
    void bset_lock_method(int);
    void bset_unlock_method(int);
    int is_locked(int);
    int get_behset();
};
```

The status of the methods in a class is stored in the integer array guard_bits. The behavior set class provides functions to lock a method, unlock a method and query the status of a method.

This paragraph explains how synchronization of method executions is achieved in ASP via behavior sets. The behavior set of the most current behavior is stored with the actor. When an actor is constructed, it sends a message to its initial behavior (invokes the method **set_parent**) to set the parent actor field in the behavior. The behavior passes back its behavior set as the return value of the message. When the initial behavior does a become, the newly created behavior computes a new behavior set. The behavior set is conveyed to the actor in the same manner. The actor stores the behavior set of its most current behavior.

An actor may attempt to invoke its behavior's method as a result of one of the following events:

a. a new message appears on the actor's mail queues;

b. a behavior does a become;

In case (a), if there is a waiting behavior, the actor checks if method is locked (Example 6.1). If the method is not locked, the message is sent to the waiting behavior. Otherwise the message gets queued until the method is unlocked (usually by the invocation of some other method of the behavior). In case (b), the new behavior informs the actor that it is

ready to process a message. The actor processes its mail queue (assuming it is not empty), finds the first authorized message and mails it to the waiting behavior (Example 6.1). This mechanism ensures the proper ordering of method executions for the behavior. Method locking via behavior sets is illustrated in the solution of the dining philosopher's problem, described in the next section.

6.6. Examples

We will describe the solution to the problem of concurrent factorial computation and the dining philosophers problem using ASP. Each problem illustrates different features of the actor system.

6.6.1. Concurrent Factorial Computation

The concurrent factorial example uses the divide and conquer technique to compute the factorial of a given number [Kafura, Lee 89b]. As shown in Figure 6.4, the **app** behavior is the driver for the concurrent factorial application. It queries the user for a number and displays the factorial of the number.

Example 6.4 shows relevant parts of the code for this example. The class **app** is the generic driver for all ASP applications. The method **fctrl** is used to compute the factorial of a number (Num) obtained from the user. The factorial actor (**fact**) is created and sent a message with Cbox handle and the number for which the factorial is to be computed. The driver class awaits the results by invoking the method **rcvfromcbox** of the Cbox.

The **fact** actor accepts request for computing factorials and forwards the work to the **prod** actor. In the method **compute_factorial** (Example 6.4), the **fact** actor creates an actor

prod. A message is sent to the actor **prod** to compute the range product for the range [1..Num]. The **fact** actor then does a become and is free to process further requests for computing factorials. The **prod** actor is responsible for returning the result directly to the client **app**.

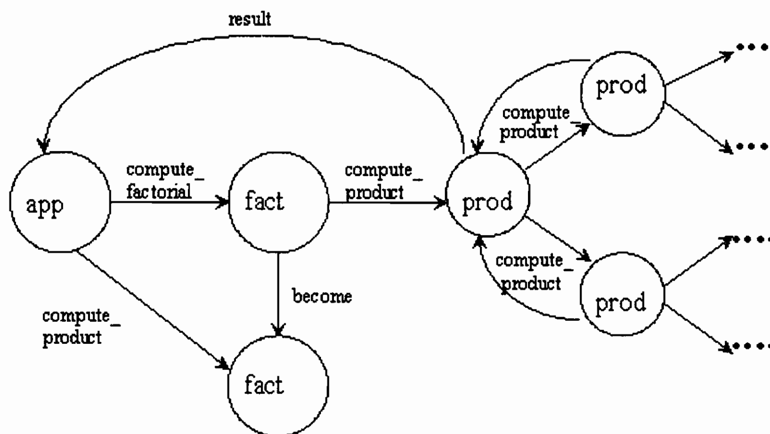


Figure 6.6 Concurrent Factorial Example

This example illustrates how inter-object and intra-object concurrency is exploited using the actor model. The **prod** actor creates multiple actors and splits the task of computing the product between them. Many **prod** actors can concurrently compute sub-products of the range product, thereby exploiting intra-object concurrency. The **fact** actor may specify a replacement behavior at the beginning of its script and continue with its work. Multiple behaviors of the **fact** actor can concurrently process request for computing factorials, thereby exploiting the concurrency within the same actor (inter-object concurrency).

EXAMPLE 6.4

DRIVER CLASS APP

```
/**********************************************************************
*
* fctrl --
*     computes the factorial for the number input by the user
**********************************************************************
/
app::fctrl(cbox *inst_cbox)
{
int number;

   inout << "What factorial would you like to compute? ";
   inout >> number;

   actor* actor_fact = New(fact());
   ((fact*)actor_fact)->compute_factorial(*(handle *)inst_cbox, number);
   int *IntPtr = inst_cbox->rcvfromcbox();
   int IntRes = *IntPtr;

   inout << "The factorial of " << number << " is " << IntRes << "\n";

}
//*************************************************************
// method that computes the factorial of a number passed to it
//*************************************************************
int fact::compute_factorial(handle inst_cbox, int number)
{
   actor *actor_prod = New (prod());
   ((prod *)actor_prod)->compute_product(inst_cbox,1,number);

   /*
     Doing a become before quitting
   */
   become (new {local} fact ());

   INT_RETURN(1);
}
//*******************************************************************
// method that computes the product of numbers in the range low to high
//*******************************************************************
int prod::compute_product(handle inst_cbox, int low, int high)
{
int RetCode, mid, sub1, sub2, *ResPtr;
actor *actor_prod1;
actor *actor_prod2;
```

```
cbox  *sub_cbox;

   if (low >= high)
      RetCode = ((cbox *)&inst_cbox)->reply2cbox(low);
   else
   {
      mid = (low+high)/2;

      actor_prod1 = New (prod());
      actor_prod2 = New (prod());

      sub_cbox = (cbox *) new {local} cbox();

      ((prod *)actor_prod1)->compute_product
                (*(handle *)sub_cbox,low,mid);
      ((prod *)actor_prod2)->compute_product
                (*(handle *)sub_cbox,mid+1,high);

      ResPtr =  sub_cbox->rcvfromcbox();
      sub1 = *ResPtr;
      ResPtr =  sub_cbox->rcvfromcbox();
      sub2 = *ResPtr;

      RetCode = ((cbox *)&inst_cbox)->reply2cbox(sub1 * sub2);
   }

   INT_RETURN(1);
}
```

The **prod** actor computes the range product by the divide-and-conquer method. In the method **compute_product** (Example 6.4), the **prod** actor splits the range into two subranges, creates two new **prod** actors and sends each of them a message to compute the product of a subrange. Each message contains a Cbox to return results. The same Cbox is passed in both the messages, since the order in which the results are returned is not important. The **prod** actor awaits the result (subproduct) from both the messages. The computation proceeds in this manner, with each **prod** actor splitting the range into sub-ranges until the range cannot be split any further. At any given stage, when the results from both messages become available, the **prod** actor returns the product of the results to the Cbox passed to it. In this manner, a sub-product is computed at each stage and

returned to the previous stage (Figure 6.4). The **prod** actor created initially computes the final result of the factorial and returns the result to the client **app**.

6.6.2. Dining Philosopher's Problem

We have modeled the classic dining philosopher's problem using the actor system (Figure 6.7). This instrumentation of the dining philosophers problem does not attempt to avoid deadlock since the only purpose of this application is to illustrate the synchronization of method executions in ASP using behavior sets.

As in the previous example, the actor **app** is the driver of the application. The driver creates actors to represent the five forks and five philosophers (Example 6.5). The identifiers for the left and right forks of each philosopher are passed to it in the constructor. The dining philosopher system is started by sending each philosopher a message to execute its **think_eat** process. A Cbox is passed in the message. When a philosopher is done executing the **think_eat** process a certain number of times (specified by **MsgTfc**), it exits by replying to the Cbox.

The **filo** actor models the actions of a philosopher (Example 6.5). In its **think_eat** method, each philosopher thinks for a random amount of time and decides to eat. It makes requests to lift its right and left forks. When the forks become available, the philosopher proceeds to eat for a random amount of time. Once the philosopher is done eating, the right and left forks are released. To continue its activity of thinking and eating, the philosopher specifies a replacement behavior and sends it a message to execute the **think_eat** method.

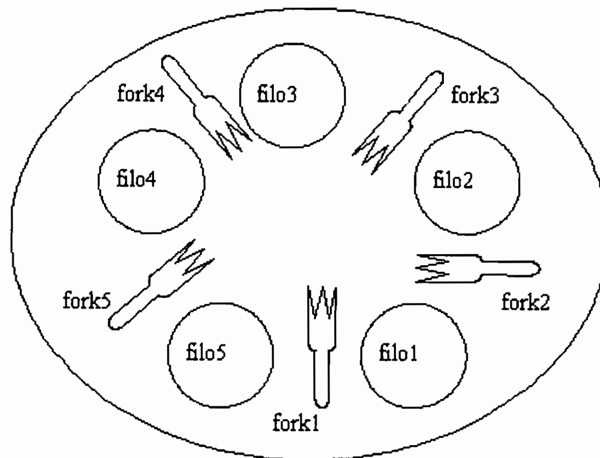


Figure 6.7 Dining Philosophers Problem

Each fork is modeled by the **fork** actor (Example 6.5). The **fork** actor has two methods: **lift** and **release**. When a philosopher makes a request to lift a fork, a message is sent to the fork actor to execute its lift method. If the fork is not being used, the lift method is executed. On the other hand, if the fork is being used, the request for lifting the fork is stored on the actor's mail queue. This synchronization is achieved via behavior sets. When a fork actor is created, it has an initial behavior where the **lift** method is open and the **release** method is closed. As explained in Section 5, an open method is a method for which the object can accept requests. Initially the fork is in a state where it may be lifted, but not released. When the fork is lifted, i.e., the **lift** method is executed, the fork actor specifies a replacement behavior in which the **lift** method is closed and the **release** method is opened. This means that after the fork is lifted, the fork actor becomes an actor which may only be released. The synchronization for the **release** method works in a similar manner.

At the time of creation, the behavior set of the **fork** actor has the **lift** method open and the **release** method closed. When the fork is lifted, it changes its state by specifying a

replacement behavior. The new behavior recomputes the behavior set, in which the **lift** method is closed and the **release** method is opened. Depending on the state the fork is in, it presents a different interface. Each time the fork changes its state, the visibility of the interface is modified through behavior sets. The state transitions of the fork actor,

released→lifted→released→.....

are modeled by the become operation (replacement behaviors). The interface of the **fork** actor in each of these states is controlled by means of behavior sets.

EXAMPLE 6.5

DRIVER CLASS APP

```
app::app()
{
     ............
     ............
     /*
       query the user for message traffic
     */
     inout << "How many msgs (per philosopher) do you want? ";
     inout >> StartVal;

     /*
       create five forks
     */
     actor_fk1 = New (fork(1,0));
     actor_fk2 = New (fork(2,0));
     actor_fk3 = New (fork(3,0));
     actor_fk4 = New (fork(4,0));
     actor_fk5 = New (fork(5,0));

     /*
       create five philosophers
     */
     actor_fil1 = New (filo(1, *(handle *)actor_fk1,
            *(handle *)actor_fk5, StartVal));
     actor_fil2 = New (filo(2, *(handle *)actor_fk2,
            *(handle *)actor_fk1, StartVal));
     actor_fil3 = New (filo(3, *(handle *)actor_fk3,
            *(handle *)actor_fk2, StartVal));
     actor_fil4 = New (filo(4, *(handle *)actor_fk4,
            *(handle *)actor_fk3, StartVal));
     actor_fil5 = New (filo(5, *(handle *)actor_fk5,
            *(handle *)actor_fk4, StartVal));

     /*
       send all of them messages
     */
     ((filo *)actor_fil3)->think_eat(*(handle *)inst_cbox);
     ((filo *)actor_fil1)->think_eat(*(handle *)inst_cbox);
     ((filo *)actor_fil5)->think_eat(*(handle *)inst_cbox);
     ((filo *)actor_fil4)->think_eat(*(handle *)inst_cbox);
     ((filo *)actor_fil2)->think_eat(*(handle *)inst_cbox);

     /*
```

```
        wait on the cbox
      */
      for (tmpVal=0;tmpVal<5;tmpVal++)
        IntPtr = inst_cbox->rcvfromcbox();

   }
   ...............
   ...............
}
```

PHILOSOPHER CLASS FILO

```
//*******************************************************************
// think and eat process of the philosopher
//*******************************************************************
int filo::think_eat(handle repbox)
{
timevalmytime;
unsigned int msectime, *IntPtr;
cbox *filo_cbox;
int RetCode;

   filo_cbox = (cbox *) new {remote} cbox();

   /*
     Wait For A Random Amount Of Time
   */
   gettimeofday (&mytime, NULL);
   msectime = (mytime.tv_sec % 10000) * fil_id;
   //inout << "msectime is " << msectime << " \n";
   while (--msectime);

   /*
     Attempt to lift forks to start eating
   */
   ((fork *)&r_f)->lift(*(handle *)filo_cbox);
   ((fork *)&l_f)->lift(*(handle *)filo_cbox);

   IntPtr = filo_cbox->rcvfromcbox();
   IntPtr = filo_cbox->rcvfromcbox();

   /*
     The philosopher has commenced eating - announce it
   */
   inout << "Philosopher " << fil_id << " is now EATING\n";

   /*
     Eat For A Random Amount Of Time
   */
   gettimeofday (&mytime, NULL);
```

```
   msectime = (mytime.tv_sec % 10000) * (fil_id + 2);
   //inout << "msectime is " << msectime << " \n";
   while (--msectime);

   /*
     Done eating - release forks now
   */
   ((fork *)&r_f)->release(*(handle *)filo_cbox);
   ((fork *)&l_f)->release(*(handle *)filo_cbox);

   IntPtr = filo_cbox->rcvfromcbox();
   IntPtr = filo_cbox->rcvfromcbox();

   /*
     The philosopher has started thinking - announce it
   */
   inout << "Philosopher " << fil_id << " is now THINKING\n";

   /*
     if we are done, respond to the cbox else send a message to myself
     and do a become
   */
   if (msg_tfc-1)
   {
     ((filo *)&get_parent())->think_eat(repbox);
     become (new {remote} filo(fil_id,r_f,l_f,--msg_tfc));
   }
   else
     RetCode = ((cbox *)&repbox)->reply2cbox(1);

   INT_RETURN(1);

}
```

CLASS FORK

```
//*****************************************************************
// simulate the fork lift and become an unliftable fork
//*****************************************************************
int fork::lift(handle inst_cbox)
{
int RetCode;

   RetCode = ((cbox *)&inst_cbox)->reply2cbox(1);
   /*
     become a fork which is ok for releasing only
   */
   become (new {local} fork(fork_no, 1));

   INT_RETURN(1);
```

```
}
//*****************************************************************
// simulate the fork release and become a liftable fork
//*****************************************************************
int fork::release(handle inst_cbox)
{
int RetCode;

  RetCode = ((cbox *)&inst_cbox)->reply2cbox(1);
  /*
    become a fork which is ok for lifting only
  */
  become (new {local} fork(fork_no, 0));

  INT_RETURN(1);
}
```

CHAPTER 7

CONCLUSIONS

7.1. Introduction

The previous chapters have discussed the problems encountered in the course of this work and the solutions devised to circumvent these problems. This chapter will discuss the experiences while integrating actor-based concurrency in a distributed, object-oriented system. The project attempts to combine the actor model of concurrency with a distributed, object-oriented system, ESP, written in C++. Both ESP and the actor model possess certain properties which made them ideal candidates for the conceptual models and implementation platforms respectively. However, each of them imposes certain restrictions, which have to be accounted for in the design of the system. Integration of actor-based concurrency and inheritance in an object-oriented domain cause certain interferences which have to be resolved.

While attempting to integrate actor-based concurrency in an object-oriented system, differences were encountered between the invocation semantics of the actor model and the invocation semantics required by inherited methods. These differences are not confined to the actor model; they arise in any situation where the invocation semantics are similar to those of the actor model. [Kafura et. al. 90] argue that they are caused by the deeper differences between the features of concurrency and inheritance. Other interferences have been observed by [Kafura, Lee 89a] and [Papathomas 89a]. A

discussion of these can be found in Chapter 6. The problem is presented and a taxonomy of solutions for it is discussed.

7.2. Class Inheritance in ESP

The actor system is built using an object-oriented paradigm. The entities in the actor model such as the actor, behavior and Cbox are modeled as objects. Communication between the objects is accomplished by message passing. ESP is a suitable platform since objects in ESP communicate by passing messages. The support for communication by message passing is built into the ESP kernel. Since the actor system has been designed as an application of ESP, all actor system objects are derived from the ESP base class. Therefore, they inherit all the features necessary to achieve communication between objects. When objects have specialized needs, for example the redirection of behavior messages to the parent actor, they can customize the available features by redefining the methods in the derived class.

In many other respects, ESP is an excellent test-bed for providing the actor abstraction. The call/return mechanism in ESP and the communication in an actor system have a similar metaphor. An object in ESP does not block when it sends a request message to another object. It is blocked only when it needs the results of the computation requested in the message. In the actor model, communication between actors is asynchronous. The actor is blocked when it attempts to evaluate its Cbox for results. ESP uses futures for synchronization and return of replies. Futures provide much of the same semantics that Cboxes are intended to provide in the actor model. However, for reasons explained in Chapter 6, the futures mechanism could not be used. Instead the reply return mechanism

was designed using Cboxes. In general, the design of the communication and call/return mechanisms in ESP make it a suitable candidate on which to build an actor platform.

As discussed in the introduction, it is possible to realize the inherent concurrency in an application on the ESP platform. The ESP kernel supports remote object creation and distributed computation. These features are provided in a manner that is transparent to the application. If the application can express its concurrency, the ESP kernel has mechanisms that can exploit it, by being able to distribute the computation.

7.3. Actor Model

The effectiveness of ASP as a platform for building concurrent applications can be ascertained by the ease of developing applications. The actor model provides the ability to express the inherent concurrency of a problem. As discussed in previous chapters, it is possible to exploit intra-object and inter-object concurrency using the actor model. Concurrent computation can be achieved by creating multiple actors and sending them messages (intra-object concurrency). Consider the concurrent factorial example discussed in Chapter 6. The work of computing the product of a range was split by creating multiple actors and sending them messages to compute sub-products. The Cbox was used to synchronize the computation of the actors and receive the results. Concurrency within the behaviors of the same actor (inter-object concurrency) can also be achieved by the actor computing the factorial doing a become at the beginning of its script. This would allow multiple factorial computations to be processed simultaneously. The example of the concurrent factorial illustrates that the actor paradigm provides a natural expression for concurrency. The concurrency inherent in a system can be easily expressed by creating multiple actors or by specifying a replacement behavior.

Using the actor metaphor, it is very easy to model a set of independent entities that communicate with each other. Consider the example of the classic dining philosophers problem. The philosophers and their activities can be very easily simulated by modeling the philosophers as actors. Multiple philosopher actors are created and execute concurrently. Each philosopher actor is an independent entity and performs actions based on messages received. The forks are represented by actors which can be either lifted or released. The state transitions of the fork actors:

lifted → released → lifted .....

are easily represented by the mechanism of specifying a replacement behavior. When an (idle) fork is lifted, it becomes a fork that may only be released and vice versa.

7.4. The Problem of Invocation Semantics

7.4.1. Description of the Problem

In the context of Hybrid, a concurrent object-oriented language, [Papathomas 89b] defines the requirements for concurrent object-oriented languages. These requirements serve to set the context for discussing the problem of invocation semantics. According to [Papathomas 89a], the "main requirement is that the concurrency features of a language be compatible with object-oriented features such as encapsulation, data abstraction and inheritance, and promote the development of applications by combining and tailoring already existing objects".

Consider two concurrent objects, Sender and Receiver (Figure 7.1). The Receiver object class has been constructed by inheritance. The method f() is defined in a super class and

the method g() in a subclass. The invocations, shown in Figure 7.1 by means of arrows, are of two types:

(a) Invocation of method f() by Sender;

(b) Invocation of method f() by its subclass g()

The invocation of method f() of the Receiver object should not cause the Sender to block. The message is sent asynchronously by the Sender, and is possibly queued at the Receiver object. The Receiver object will accept the message based on its message selection policy.

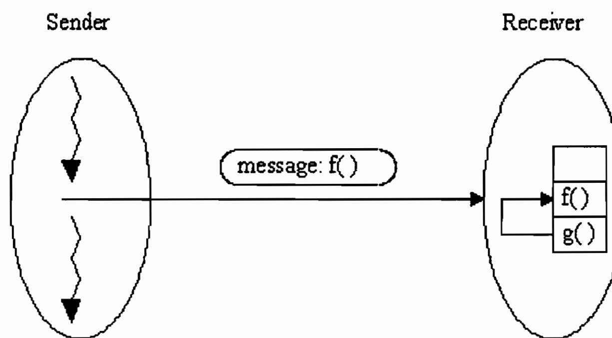


Figure 7.1 Two Forms Of Invocation

The invocation of method f() by the method g() should cause g() to block. The invocation is synchronous and does not cause a message to be sent to the mail queue of the Receiver object. Invocations within an object, such as sub-class and super-class invocations, are encapsulated within the object. When the method f() finishes its execution, it causes method g() to be unblocked and continue with its execution.

The invocation semantics discussed for case (b) are the correct semantics for invocations within a class. Systems like ESP, which provide mutual exclusion, do not allow multiple methods of an object to execute simultaneously. Without localizing the sub-class and super-class method invocations, these systems could deadlock. Consider the scenario where the invocation of f() by g() causes a message to be sent from the Receiver object to itself. Suppose the method g() is blocked, waiting on the results of method f(). The method f() will not be invoked until the Receiver object is unblocked. However, the Receiver object will not be unblocked until method f() is invoked.

The semantics for local and remote invocations have been stated. A system which supports both styles of invocations should be designed with the following objectives:

- Invocation Transparency: a method is invoked without regard for its implementation;
- Implementation Transparency: a method is implemented without regard for how it will be invoked.

These objectives are compatible with the design requirements stated by [Papathomas 89b]. The problem encountered is in achieving implementation transparency. The local and remote invocation semantics make different assumptions about the reply return mechanism. To reply to method invocations by the Sender object, the Receiver object assumes that the means of reply are passed as part of the message. For example, a handle to the Cbox object is passed to the invoked object so that it may reply using the Cbox. On the other hand, in a local invocation, the object would assume that the reply is returned implicitly.

7.4.2. A Taxonomy of Solutions

Figure 7.2 illustrates the possible strategies for solving the problem of invocation semantics. These strategies can be broadly divided into those that pass a reply point (such as a Cbox) explicitly to the object and those that manage the reply point implicitly.

The implicit reply return mechanism in ESP is discussed to illustrate how ESP addresses this problem. With every request message in ESP, a hidden structure (future) is passed. The future is used by the underlying mechanism to format a reply and send it to the appropriate destination. In the case of a local invocation, the future structure is not passed and a local return is assumed.

The main drawback of using futures is that the return path has to be maintained. Consider the factorial application described in the previous chapter. The **fact** actor forwards the computation of the range product to the **prod** actor and plays no part in the computation thereafter. The results are returned directly to the client actor by the **prod** actor. The above semantics cannot be implemented using the futures mechanism. In a similar application written in ESP, the **fact** actor would have to wait on the result computed by the **prod** actor and return it to the client. There is a potentially large expense to be paid for maintaining the return path and saving the context of the **fact** actor. To implement the desired semantics in ESP, an explicit future would have to be passed to the **fact** actor.

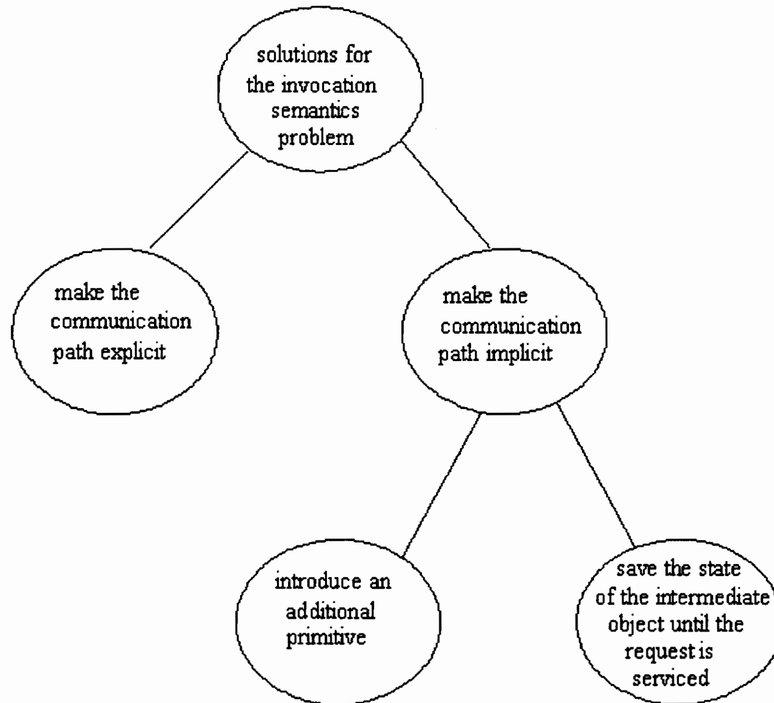


Figure 7.2 Invocation Semantics Problem: Taxonomy of Solutions

One possible solution would be to introduce a primitive in the language which would permit the **fact** actor to associate the future identifier of the client with the message passed between the **fact** and **prod** actors. The **fact** actor would not have to remain in the return path until the computation was finished. However, the implementor of the method in the **fact** class would have to know if the method was invoked remotely (forwarding possible) or locally (forwarding may not be possible).

The other strategy involves passing an explicit reply point, such as a Cbox (discussed in Chapter 6, Section 4) to the object. In the concurrent factorial example, the **fact** actor could pass the reply point on to the **prod** actor, which could reply directly to the client. The drawback in this strategy is that when the method of the object is locally invoked, it

is still necessary to pass the reply point. The implementor of the method has to perform additional checking on the reply point to determine how to return the result. For instance, in ASP, if the Cbox handle passed is NULL, the object would perform a local return.

For reasons discussed in the previous chapter, the decision was made to use explicit reply points in the communications of actors in ASP. All of the possible strategies discussed have certain advantages and drawbacks. The choice of a solution would depend on the design requirements of the system.

7.5. The Problem of Acquaintances

One of the needs in an actor system is an automatic garbage collection scheme [Kafura 88] to remove the burden of managing memory resources from the developer, while unifying process and memory management. To determine if an actor can be garbage collected or not, the garbage collection algorithm needs to maintain a list of acquaintances for each actor. We assume that there is an acquaintance manager at each node of the actor system, which performs these tasks. Acquaintances are acquired by an actor by receiving an identifier for the acquaintance in a request message. Whenever an acquaintance is sent in a message to the actor, it is the responsibility of the actor system to inform the acquaintance manager about a new acquaintance relationship. We will describe the problem we encountered in accomplishing this objective.

The problem is to identify when an acquaintance is being passed in a request message. This information is available in the method which creates and passes the request message. However, the actor system gets control of the message only when the overloading of the pointer-to-member-function method of the behavior class is invoked (refer Chapter 6). At

this time, the arguments passed to the method are formatted into a message and it is not possible to identify the acquaintances passed.

Several solutions were considered. C++ provides a copy constructor [Stroustroup 92] which is invoked when an object is passed as an argument to a function. However, the problem with this approach is that the copy constructor is invoked whenever the object is copied, such as in an assignment. It is not possible to isolate the case when the object was being passed in an argument list. Another approach is to use a separate primitive (like send_acquaintance) to pass acquaintances. Using the definition of the overloaded pointer-to-member-function operator (refer Chapter 6), it is possible to identify every acquaintance being passed and inform the acquaintance manager accordingly. However, this solution imposes considerable programming burden on the developer. The solution to the problem of maintaining global acquaintance information for garbage collection is a possible future extension of the present work.

BIBLIOGRAPHY

[Agarwal, Arvind 82] Agarwal T., Arvind, "Data Flow Systems", Computer, Volume 15, Number 2, February 1982.

[Agha 86a] Agha G., ACTORS: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, 1986.

[Agha 86b] Agha G., "An Overview of Actor Languages", SIGPLAN Notices, Volume 21 Number 10, October 1986.

[Agha, Hewitt 87] Agha G., Hewitt C., "Concurrent Programming using Actors", Object-Oriented Concurrent Programming, (ed.) Yonezawa, A., Tokoro, M., MIT Press, 1987, 37-53.

[Athas, Boden 88] Athas W., Boden N., Cantor: An Actor Programming System for Scientific Computing", Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, 1988, SIGPLAN Notices, Volume 24, Number 4, April 1989.

[Backus 78] Backus, J., "Can programming be liberated from the von neumann style? a functional style and its algebra of programs", Volume 21, Number 8, Pages 613-641, August 1978.

[Black, et. al. 87] Black A., Hutchinson N., Jul E., Levy H., Carter L., "Distribution and Abstract Types in Emerald", IEEE Transactions on Software Engineering, Volume SE-13, Number 1, January 1987.

[Briot 88] Briot J.-P., de Ratuld J., "Design of a distributed implementation of ABCL/1", SIGPLAN Notices, Volume 24, Number 4, Pages 15-17, ACM SIGPLAN Workshop on Object-Based Concurrent Programming, San Diego, CA, September 1988.

[Briot 89] Briot J.-P.,, "Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment", Proceedings of the Third European Conference on Object-Oriented Programming, 1989.

[Campbell, et. al. 87] Campbell R., Johnston G., Russo V., "Class Hierarchical Open Interface for Custom Embedded Systems", Operating Systems Review, Volume 23, Number 3, July 1987, Pages 9-17.

[Chatterjee 89] Chatterjee A., "Futures: A Mechanism for Concurrency Among Objects", MCC Technical Report Number: ACT-ESP-225-89, June 1989.

[Chatterjee et. al. 89] Chatterjee A., Khanna A., Hung Y., McLaren R., Narayanswamy S., Ajmani N., "ES-Kit: A Distributed, Object-Oriented Operating System", MCC Technical Report Number: ACT-ESP-374-89, October 1989.

[Chin, Chanson 91] Chin R., Chanson S., "Distributed Object-Based Programming Systems", ACM Computing Surveys, Volume 23, Number 1, March 1991.

[Clinger 81] Clinger W., Foundations of Actor Semantics, Technical Report 633, MIT Artificial Intelligence Laboratory, 1981.

[Detlefs et. al. 88] Detlefs D., Herlihy M., Wing J., "Inheritance of Synchronization and Recovery Properties in Avalon/C++", IEEE Computer Magazine, Pages 57-69, December 1988.

[Gentlemen 85] Gentlemen W., Using the Harmony Operating System, National Research Council of Canada Report No. 24685 national Research Council of Canada, Ottawa, Canada, May 1985.

[Hoare 78] Hoare C.A.R., "Communicating Sequential Processes", CACM, Volume 21, Number 8, August 1978, Pages 666-677.

[Henderson 80] Henderson P., Functional Programming Application and Implementation, Prentice-Hall International Series in Computer Science, C.A.R. Hoare Series Editor, 1980.

[Herlihy, Liskov 82] Herlihy M., Liskov B., "A value transmission method for abstract data types", ACM Transactions of Programming Language Systems, Volume 4, October 1982, Pages 527-551.

[Kafura 88] Kafura D., "Concurrent Object-Oriented Real-Time Systems Research", Technical Report, TR 88-47, Department of Computer Science, Virginia Polytechnic Institute and State University, 1988.

[Kafura, et. al. 90] Kafura D., Lavender G., Joshi N., "Recent Progress and Problems in Combining Actor-Based Concurrency with Object-Oriented Programming", Workshop on Object-Based Concurrency at OOPSLA 1990, October 20, 1990.

[Kafura, Lee 89a] Kafura D., Lee K., "Inheritance in Actor-Based Concurrent Object-Oriented Languages", Proceedings of the Third European Conference on Object-Oriented Programming, 1989, Pages 131-145.

[Kafura, Lee 89b] Kafura D., Lee K., "ACT++: Building a Concurrent C++ with Actors", Journal of Object-Oriented Programming, May 1990.

[Khanna 89] Khanna A., "Class Management in the Experimental Systems Domain", MCC Technical Report Number: ACT-ESP-305-89, August 1989.

[Leddy et. al. 88] Leddy W., Virmani V., Widjaja T., "ES-Kit Kernel Release I Design Notes", Technical Report:ACA-ESP-141-88, Microelectronics and Computer Technology Corporation, April 1988.

[Leddy, Smith 89] Leddy W., Smith K., "The Design of the Experimental Systems Kernel", Proceedings of the Conference on Hypercube and Concurrent Computer Applications, 1989, Monterey CA.

[McAffer, Berry 89] McAffer J., Berry B., "Actra A Multitasking/Multiprocessing Smalltalk", Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, 1988, SIGPLAN Notices, Volume 24, Number 4, April 1989.

[Milner 80] Milner R., A Calculus of Communicating Systems, Springer-Verlag, 1980.

[Molloy 82] Molloy M., "Performance Analysis Using Stochastic Petri Nets", IEEE Transactions on Computers, Volume C-31, Number 9, Pages 913-917, September 1982.

[Nierstrasz 88] Nierstrasz O., "A Survey of Object-Oriented Concepts", Object-Oriented Concepts and Databases, (ed.) Kim and Lochovsky, Addison-Wesley, 1988.

[Papathomas 89] Papathomas M., "Concurrency Issues in Object-Oriented Languages", Object Oriented Development, (ed.) D. Tsichritzis, Pages 207-245, University of Geneva, 1989.

[Papathomas 89] Papathomas M., "Integrating Concurrency and Object-Oriented Programming: An Evaluation of Hybrid", Object Management, (ed.) D. Tsichritzis, Pages 229-244, University of Geneva, 1990.

[Smith 88] Smith K., "ES-Kit Software Development Environment The ISSD and Clients", MCC Technical Report Number: ACT-ESP-140-88, April 1988.

[Smith 89] Smith K., Smith R., Caldwell G., Porter C., Leddy W., Khanna A., Chatterjee A., Hung Y., Hahn D., Allen W., "Experimental Systems Project at MCC", MCC Technical Report Number: ACT-ESP-089-89, March 1989.

[Smith 90] Smith K., "ES-Kit Distributed Kernel Principles of Operation", MCC Technical Report Number: ACT-ESP-90, May 1990.

[Stroustroup 92] Stroustroup B., The C++ Programming Language Second Edition, Addison-Wesley, Menlo Park, CA, 1991.

[Surma 90] Surma E., "Extensible Software Platform Software Development Environment Programmer's Guide", MCC Technical Report Number: ACT-ESP-223-88, August 1990.

[Tanenbaum, Renesse 85] Tanenbaum A., Renesse R., "Distributed Operating Systems", Computing Surveys, Volume 17, Number 4, December 1985.

[Theriault 83] Theriault D., Issues in the Design and Implementation of Act2, Technical Report 728, MIT Artificial Intelligence Laboratory, 1983.

[Tomlinson, Singh 89] Tomlinson C., Singh V., "Inheritance and Synchronization with Enabled-Sets", OOPSLA 1989, Conference Proceedings Special Issue of SIGPLAN Notices, Volume 24, Number 10, October 1989, Pages 103-112.

[Tomlinson, Scheevel 87] Tomlinson C., Scheevel M. "Concurrent Object-Oriented Programming Languages", Object-Oriented Concurrent Programming, (ed.) Yonezawa A., Tokoro M., MIT Press, 1987.

[Wegner 87] Wegner P., "Dimensions of Object-Based Language Design", Proceedings of OOPSLA 1987, SIGPLAN Notices 22:12, 1987, Pages 168-182.

[Yokote, Tokoro 86] Yokote Y., Tokoro M., "Concurrent Programming in Concurrent Smalltalk", Object-Oriented Concurrent Programming, (ed.) Yonezawa A., Tokoro M., MIT Press, 1987.

[Yonezawa et. al. 87] Yonezawa A., Shibayama E., Takada T., Honda Y., "Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1", Object-Oriented Concurrent Programming, (ed.) Yonezawa A., Tokoro M., MIT Press, 1987, Pages 55-89.

[Walker et. al. 90] Walker E., Floyd R., Neves P., "Asynchronous Remote Operation Execution in Distributed Systems", IEEE Proceedings of 10th International Conference on Distributed Computing Systems, 1990, Pages 253-259.

APPENDIX A

ASP System Classes

```
// ----------------------------------------------------------------------
// actor.h
// *******************************************************************
// this file includes the definition of the actor class
// its private parts and its methods
//  author: nandan joshi
//  project: actors on eskit
// *******************************************************************

#include "future.h"
#include  "actor_req.h"
#include  "behset.h"
#include  "behavior.h"

#define   TRUE 1
#define   FALSE  0
#defineMETH_OFFSET 2147483648

#define New(beh)  new {anywhere} actor( *(handle *)new {anywhere}beh)

#ifndef ACTOR
#define ACTOR
class actor : public remote_base
{
  actor_req*  mail_q_ptr;        /* pointer to actor's mail queue*/
  bool        waiting_behavior;  /* is there a waiting behavior  */
  handle      most_curr_behavior;/* handle to waiting behavior     */
  handle      curr_behset;      /* behavior set handle of the mcb*/
protected:
  local_streams inout;
public:
  actor(handle);        // constructor for class actor
  ~actor();                   // destructor for class actor
  __future_ref__ receive_req_msg(int , int ,...);
  int deliver_req_msg(handle);
  void add_actor_req(request_msg*, int);
  request_msg * get_actor_req();
   int update_curr_beh(handle);
};
#endif

//*******************************************************************
```

```
// -----------------------------------------------------------------
//        actor.cc
// -----------------------------------------------------------------
// ****************************************************************
//
//   This file includes the definitions of the methods of class actor.
//
//*****************************************************************

#include "def.h"
#include "utils.h"
#include "esk_base.h"
#include "msg_stuff.h"
#include "post_off.h"
#include "future.h"
#include "k_fun.h"
#include "node_kernel.h"

#include "actor.h"
#include "interm.h"
#include "behset.h"
#include "behavior.cc"

 //****************************************************************
 //  Constructor for the actor class:-
 //     This user invokes the constructor of a behavior by New(beh)
 //     which gets expanded to new actor(new beh); The new operation
 //     on the behavior returns a handle to the behavior and the
 //     actor's constructor gets invoked with the behavior handle
 //     passed to it as an argument
 //***************************************************************
actor::actor(handle init_beh) : inout(this)
{
  interm* interm_inst;

  mail_q_ptr = NULL;
  waiting_behavior = TRUE;
  most_curr_behavior = init_beh;

  interm_inst = (interm *) &init_beh;
  curr_behset = interm_inst->set_parent(*(handle *)this);

  CONS_RETURN;
}

//****************************************************************
//Destructor for the actor class
//****************************************************************
actor::~actor() { }

//****************************************************************
```

```
// Receive_req_msg:
//   This method is invoked by the receiver's node kernel. If
//   a waiting behavior exists for the actor, then the msg is
//   sent to the behavior, else it is queued up on the
//   actor's queue.
//*****************************************************************

__future_ref__ actor::receive_req_msg(int meth_id, int arg_len, ...)
{
  behset *curr_beh_ptr;

  if (waiting_behavior)
  {
     /*
    then send the request to it
     */
     request_msg* pmsg  = new request_msg (&most_curr_behavior,meth_id,
                                arg_len,(int*)&arg_len+1,NULL);

     /*
    Check if the method is locked or not before mailing it
     */
     if (((behset *)&curr_behset)->is_locked(meth_id+METH_OFFSET))
       add_actor_req(pmsg, meth_id);
     else
     {
       waiting_behavior = FALSE;
       POST_OFFICE->mail(pmsg);
     }
  }
  else
  {
     /*
     create a request_msg and queue it up on the actor's mail queue
     */
     request_msg* pmsg=new request_msg(&most_curr_behavior,meth_id,
                      arg_len, (int*)&arg_len+1,NULL);
                      add_actor_req(pmsg, meth_id);
     pmsg = NULL;
  }
}    /* end of function receive_req_msg  */


//******************************************************************
//Deliver_req_msg:
// This method is invoked by a newly created behavior (as a result
// of the become operation). The behavior is passed a request_msg
// if one exists and the actor's most_curr_beh info is updated
// OR the actor simply records the fact that there is a
// most_curr_beh and its handle is stored in waiting_behavior
//******************************************************************
```

```
int actor::deliver_req_msg(handle beh_ptr)
{
  request_msg* pmsg;

  update_curr_beh(beh_ptr);

  /*
  Go through the message queue and try to find a viable message,
  i.e., one that is not for a locked method
  If there is such a message on the actor's queue, send it to the
  newly created behavior
  */
  if (pmsg = get_actor_req())
  {
    waiting_behavior = FALSE;

    pmsg->node_id = most_curr_behavior.node_id;
    pmsg->appl_id = most_curr_behavior.appl_id;
    pmsg->clas_id = most_curr_behavior.clas_id;
    pmsg->inst_id = most_curr_behavior.inst_id;
    POST_OFFICE->mail(pmsg);
  }
  else
    waiting_behavior = TRUE;
}

//*****************************************************************
// Add_actor_req:
//   This method is called by receive_req_msg() to add a
//   request_msg to the actor's mail queue.
//*****************************************************************

void actor::add_actor_req(request_msg* req_msg, int meth_id)
{
actor_req* req = new actor_req(req_msg, meth_id);

  if ( mail_q_ptr == NULL )
    mail_q_ptr = req;
  else
  {
    actor_req* temp = mail_q_ptr;
    while (temp->next != NULL)
      temp = temp->next;
    temp->next = req;
    req->next = NULL;
  }
}   /* end of function add_actor_req()  */

//*****************************************************************
// Get_actor_req:
```

```
// This method is called by deliver_req_msg() to get a
// request_msg from the actor's mail queue.
//*****************************************************************

request_msg* actor::get_actor_req()
{
actor_req *temp, *prevtemp, *antemp;
short iter=0;

    request_msg* result = NULL;
    if ( mail_q_ptr )
    {
     /*
     Go through the mail queue to find a message which is invoking an
     "open" method; open - legal to invoke currently
     */
     prevtemp = temp = mail_q_ptr;
     while ( (temp) &&
(((behset *)&curr_behset)->is_locked((temp->msg_meth_id)+METH_OFFSET)))
     {
       prevtemp =  temp;
       temp    = temp->next;
     }
     if (temp)
     {
       if (prevtemp==temp)
         mail_q_ptr = temp->next;
       else
         prevtemp->next = temp->next;
       result = temp->message;
     }
    }
    return( result );
}    /* end of function get_actor_req()  */
//*******************************************************************
//   update_curr_beh(handle) :
// When a behavior does a become,become invokes deliver_req_msg which
// invokes this method to update  the actor's most current behavior and
// call set_parent to update the behavior's parent
//*******************************************************************
int actor::update_curr_beh(handle new_beh)
{
  interm* interm_inst;

  /*
    Set the actor's private members
  */
  most_curr_behavior = new_beh;

  /*
    Set the most current behavior's parent field
```

```
   */
   interm_inst = (interm *) &new_beh;
   curr_behset = interm_inst->set_parent(*(handle *)this);

   INT_RETURN(1);
}
// ---------------------------------------------------------------
// behavior.h
// ***************************************************************
// this file includes the definition of the behavior class
// its private parts and its methods
//  author: nandan joshi
//  project: actors on eskit
// ***************************************************************

#include "future.h"
#include "interm.h"

#ifndef BEHAVIOR
#define BEHAVIOR
class behavior : public interm
{
  local_streams inout;
public:
  behavior();               // constructor for class behavior
  ~behavior();              // destructor for class behavior
  __future_ref__ operator->()(int,int,...);
  int become(handle);
};
#endif
// ---------------------------------------------------------------
//       behavior.cc
// ***************************************************************
//       author: nandan joshi
//       project: actors on eskit
// ***************************************************************
//
//  ABSTRACT:
//  This file includes the definitions of the methods of class behavior.
//
//  CONSTRUCTOR:
//  This is invoked when an application requests creation of an actor
//  thru the creation of a behavior. Only the initial behavior for an
//  actor is created in this manner. The rest of the behaviors get
//  created by the become operation performed by the behaviors.
//  The constructor of the behavior invokes the new operation for the
//  actor, which returns a handle to it. Using this handle it binds the
//  actor to its parent.
//
//  DESTRUCTOR:
//
```

```
//  BECOME:
//  Whenever a behavior wants to specify a replacement, it does the
//  become operation. This operation causes a new behavior to be created
//  and the the method deliver_req_msg of actor to be executed.
//
// ******************************************************************

#include "utils.h"
#include "esk_base.h"
#include "post_off.h"

#include "behavior.h"
#include "actor.h"
#include "interm.cc"

//*******************************************************************
//Constructor for class behavior :
//   Returns a pointer to itself
//*******************************************************************
behavior::behavior() : inout(this)
{
  CONS_RETURN;
} /* end of constructor function for behavior   */

//*******************************************************************
//Destructor for class behavior
//*******************************************************************
behavior::~behavior()
{
}

//*******************************************************************
//Operator->():
//   Overloading of -> operator redefined for class behavior
//*******************************************************************

__future_ref__ behavior::operator->()(int meth_id, int arg_len, ...)
{
   request_msg* pmsg;
   int new_arg_len = arg_len + sizeof(int) + sizeof(int);
   pmsg = new request_msg ((handle*)this,
     (int)&actor::receive_req_msg, new_arg_len,(int*)&meth_id,NULL);
   return ((__future_ref__)pmsg);

}         /* end of operator->() overloading    */

/******************************************************************
  BECOME
******************************************************************/
int behavior::become (handle new_beh)
{
```

```
  ((actor *) &get_parent())->deliver_req_msg(new_beh);
  INT_RETURN (1);
}
// -----------------------------------------------------------------
// behset.h
// ****************************************************************
// this file includes the definition of the behset class
// its private parts and its methods
//  author: nandan joshi
//  project: distributed actors on eskit
// ****************************************************************
#ifndef BEHSET
#define BEHSET
class behset: public remote_base
{
  local_streams inout;
  int guard_bits[32];
public:
  behset();              // empty constructor for class behset
  ~behset();          // destructor for class behset
  void bset_lock_method( int );
  void bset_unlock_method( int );
  int is_locked ( int );
  int get_behset();
};
#endif
/*****************************************************************

  BEHAVIOR.CC
  author: nandan joshi
  project: actors on eskit

  ABSTRACT:
  This file includes the definitions of the methods of class behset.

  CONSTRUCTOR:

  DESTRUCTOR:

*****************************************************************/

#include <varargs.h>

#include "utils.h"
#include "esk_base.h"
#include "post_off.h"

#include "behset.h"

/*****************************************************************
  Constructor for class behset :
```

```
     Sets all guard_bits to zeroes
**********************************************************************/

behset::behset() : inout(this)
{
int LoopVar=0;

  while (LoopVar < 32)
    guard_bits[LoopVar++] = 0;

  CONS_RETURN;

} /* end of empty constructor function for behset   */

/**********************************************************************
  Destructor for class behset
**********************************************************************/

behset::~behset()
{
}
/*********************************************************************
  bset_lock_method
**********************************************************************/
void behset::bset_lock_method( int a )
{
  guard_bits[a] = 1;
  VOID_RETURN;
}

/*********************************************************************
  bset_unlock_method
**********************************************************************/
void behset::bset_unlock_method( int a )
{
  guard_bits[a] = 0;
  VOID_RETURN;
}

/**********************************************************************
  is_locked
**********************************************************************/
int behset::is_locked( int a )
{
  int i;

  INT_RETURN (guard_bits[a]);
}

/*********************************************************************
  get_behset
```

```
**********************************************************************/
int behset::get_behset()
{
  return ( guard_bits[0] );
}

// ----------------------------------------------------------------------
// cbox.h
// *******************************************************************
// this file includes the definition of the cbox class
// its private parts and its methods
//  author: nandan joshi
//  project: actors on eskit
// *******************************************************************
#ifndef CBOX
#define CBOX

typedef struct Result_S {
  void*  ResPtr;
  int  ResSize;
  struct Result_S *Next;
} Result_T;

typedef Result_T * ResultPtr;

class cbox : public remote_base
{
  local_streams inout;
  Result_T      *ResultStrPtr;
  int          ResIndex;

public :
  cbox();
  ~cbox();
  __future_ref__ operator->()(int,int,...);
  int reply2cbox(int,...);
  void* rcvfromcbox();
};
#endif

#define PTR_MORERET(x,y)\
    {void* _x = (void*)(x);\
    if (_x) { _ans_len = y;\
         answer.ptr = (_x) ;\
    } else this->r_return((void*)0);\
    return(_x);}
//***********************************************************
//   definition of class cbox
//   this class is derived from class remote_base
//***********************************************************
```

```
#include "utils.h"
#include "esk_base.h"
#include "msg_stuff.h"

#include "cbox.h"

//*****************************************************************
//   constructor for class cbox;
//*****************************************************************
cbox::cbox() : inout(this)
{
  ResIndex = -1;
  ResultStrPtr = NULL;

  lock_method((int)&cbox::rcvfromcbox);

  CONS_RETURN;
}

//*****************************************************************
// destructor for class cbox
//*****************************************************************
cbox::~cbox()
{
}

//*****************************************************************
// operator overloading for cbox class
//   defined so that we can find the length of the arguments passed
//   back as the results
//*****************************************************************
__future_ref__ cbox::operator->()(int meth_id, int arg_len, ...)
{
  int new_arg_len = arg_len + sizeof(int);
  request_msg* pmsg
  = new request_msg
(handle*)this,meth_id,new_arg_len,(int*)&arg_len,NULL);
  return ((__future_ref__) pmsg);

} /* end of operator->() overloading  */

//*****************************************************************
// reply2cbox  : copies the results into its structure and
//             unlocks rcvfromcbox
//*****************************************************************

int cbox::reply2cbox(int arg_len, ...)
{
  ResIndex++;

  Result_T    *TmpPtr;
```

```
   /*
     Calculate the number of words from the number of bytes
    */
   int norm_arg_len = (arg_len + 3) >> 2;

   /*
    Fill in the result structure
    */


   if (ResultStrPtr)
   {
     TmpPtr      = ResultStrPtr;
     while (TmpPtr->Next)
       TmpPtr = TmpPtr->Next;
     TmpPtr->Next   = (Result_T *) __builtin_new ( sizeof (Result_T) );
     TmpPtr = TmpPtr->Next;
     TmpPtr->ResPtr = (void *) __builtin_new ( norm_arg_len );
     TmpPtr->ResSize= arg_len;
     TmpPtr->Next   = NULL;
     wordcopy ( &arg_len+1, TmpPtr->ResPtr, norm_arg_len );
   }
   else
   {
     ResultStrPtr   = (Result_T *) __builtin_new ( sizeof (Result_T) );
     ResultStrPtr->ResPtr = (void *) __builtin_new ( norm_arg_len );
     ResultStrPtr->ResSize= arg_len;
     ResultStrPtr->Next = NULL;
     wordcopy ( &arg_len+1, ResultStrPtr->ResPtr, norm_arg_len );
   }

   unlock_method((int)&cbox::rcvfromcbox);

   INT_RETURN(1);
}

//*****************************************************************
// rcvfromcbox : return the 1st result in the list and delete it
//               from the list
//*****************************************************************
void* cbox::rcvfromcbox()
{
   Result_T    *TmpPtr;

   /*
   *  Store the 1st result structure and delete it from the list
   */

   TmpPtr      = ResultStrPtr;
   if (ResultStrPtr)
```

```
      ResultStrPtr   = ResultStrPtr->Next;

   /*
    * If there are no more results in the list lock this method
   */

   if (--ResIndex < 0)
      lock_method((int)&cbox::rcvfromcbox);

   PTR_MORERET (TmpPtr->ResPtr, TmpPtr->ResSize);

}

// -----------------------------------------------------------------
// interm.h
// ****************************************************************
// this file includes the definition of the interm class
// its private parts and its methods
//  author: nandan joshi
//  project: ACT/KIT
// ****************************************************************

#include "future.h"
#include "behset.h"

#ifndef INTERM
#define INTERM
class interm : public remote_base
{
  local_streams inout;
  handle        parent;    // handle to the actor for the interm
protected:
  handle      my_behset; // handle to its behavior set object
public:
  interm();                        // constructor for class interm
  ~interm();             // destructor for class interm
  handle set_parent( handle );  // returns behset *
  handle get_behset();
  handle get_parent();
};
#endif
// -----------------------------------------------------------------
//        interm.cc
// ****************************************************************
//        author: nandan joshi
//        project: ACT/KIT
// ****************************************************************
//
//  ABSTRACT:
//  This file includes the definitions of the methods of class interm.
//
```

```
//   DESTRUCTOR:
//
//   SET_PARENT:
//
// *****************************************************************
#ifndef INTERM_CC
#define INTERM_CC

#include "utils.h"
#include "esk_base.h"
#include "post_off.h"

#include "interm.h"
#include "actor.h"
#include "behset.h"

//*****************************************************************
//Constructor for class interm :
//   Returns a pointer to itself
//*****************************************************************
interm::interm() : inout(this)
{
  my_behset =  * (handle *) new {local} behset ();
  CONS_RETURN;
} /* end of constructor function for interm  */

//*****************************************************************
//Destructor for class interm
//*****************************************************************
interm::~interm()
{
}

//*****************************************************************
//      set_parent :
//*****************************************************************
handle interm::set_parent(handle parent_actor)
{
  this->parent = parent_actor;
  PTR_RETURN (&my_behset);
}

//*****************************************************************
//      get_parent :
//*****************************************************************
handle interm::get_parent()
{
  return parent;
}
//*****************************************************************
//      get_behset :
```

```
//*****************************************************************
handle interm::get_behset()
{
  return my_behset;
}
#endif
```

APPENDIX B

ASP Application Classes

```
// This application is used to launch the factorial, behavior set and
   dining philosopher applications

#define MAX_2_DIGIT_NUMBER 99
#define MAX_3_DIGIT_NUMBER 999
#define MAX_4_DIGIT_NUMBER 9999
#define MAX_5_DIGIT_NUMBER 99999

#ifndef APP
#define APP
class app : public behavior
{
   local_streams inout;
public :
   app();
   ~app();
   done();
   tbs(actor *, cbox *);
   fctrl(cbox *);
};
#endif

/***********************************************************************
*
//   Module Name : app.cc
//   Purpose    : Definition of class app
// Constructor of class app is the driver for the  application
***********************************************************************
/

#include "utils.h"
#include "esk_base.h"
#include "handle.h"

#include "inc.h"
#include "dec.h"
#include "fact.h"

#include "behset.h"
#include "actor.h"
```

```
#include "cbox.cc"      //incl because of ->() defn in cbox
#include "behavior.cc"    //incl because need to incl supclass .cc file

#include "app.h"
#include "fork.h"
#include "filo.h"

/**********************************************************************
*
  constructor for class app
**********************************************************************
/
app::app() : inout(this)
{
  int retnumber,i, finish_app, IntChoice, StartVal, *IntPtr, tmpVal;
  actor *actor_inc;
  actor *actor_ttor;

  actor *actor_fil1;
  actor *actor_fil2;
  actor *actor_fil3;
  actor *actor_fil4;
  actor *actor_fil5;

  actor *actor_fk1;
  actor *actor_fk2;
  actor *actor_fk3;
  actor *actor_fk4;
  actor *actor_fk5;

  /*
    Create an object of class cbox
  */
  cbox* inst_cbox = (cbox*) new {remote} cbox();

  inout << "You have entered the world of actors...
            the world is a stage and all that\n";
  inout << "Would you like to run the factorial(1) or
            behavior set(2) or dining philosophers (3) application? ";
  inout >> IntChoice;

  if (IntChoice == 1)
    fctrl(inst_cbox);
  else if (IntChoice == 2)
  {
    /*
      The following call causes these methods to get executed:
      Constructor of behavior
      Constructor of interm - creation of behavior set
```

```
    Constructor of actor   - set its most_curr_behavior
           invoke method set_parent of the same
    set_parent(mcb)-sets parent actor and returns behavior set
  */
  inout << "Please enter the starting value: ";
  inout >> StartVal;
  actor_inc = New(inc(StartVal));
  tbs (actor_inc, inst_cbox);
}
else
{
  /*
    query the user for message traffic
  */
  inout << "How many msgs (per philosopher) do you want? ";
  inout >> StartVal;

  /*
    create five forks
  */
  actor_fk1 = New (fork(1,0));
  actor_fk2 = New (fork(2,0));
  actor_fk3 = New (fork(3,0));
  actor_fk4 = New (fork(4,0));
  actor_fk5 = New (fork(5,0));

  /*
    create five philosophers
  */
  actor_fil1 = New (filo(1, *(handle *)actor_fk1,
        *(handle *)actor_fk5, StartVal));
  actor_fil2 = New (filo(2, *(handle *)actor_fk2,
        *(handle *)actor_fk1, StartVal));
  actor_fil3 = New (filo(3, *(handle *)actor_fk3,
        *(handle *)actor_fk2, StartVal));
  actor_fil4 = New (filo(4, *(handle *)actor_fk4,
        *(handle *)actor_fk3, StartVal));
  actor_fil5 = New (filo(5, *(handle *)actor_fk5,
        *(handle *)actor_fk4, StartVal));

  /*
    send all of them messages
  */
  ((filo *)actor_fil3)->think_eat(*(handle *)inst_cbox);
  ((filo *)actor_fil1)->think_eat(*(handle *)inst_cbox);
  ((filo *)actor_fil5)->think_eat(*(handle *)inst_cbox);
  ((filo *)actor_fil4)->think_eat(*(handle *)inst_cbox);
  ((filo *)actor_fil2)->think_eat(*(handle *)inst_cbox);

  /*
    wait on the cbox
```

```
      */
      for (tmpVal=0;tmpVal<5;tmpVal++)
         IntPtr = inst_cbox->rcvfromcbox();

   }
   done();
   CONS_RETURN;

}

/*********************************************************************
*
  destructor for class app
*********************************************************************
/
app::~app() { }

/*********************************************************************
*
* tbs --
*    test for behavior sets
*    calls to incpriv or decpriv are not waited upon by a return value
*    the reason being, that since we are redirecting the call through
the
*    ->() mechanism, eskit thinks that the response has arrived much
*    earlier
*********************************************************************
/
app::tbs(actor *actor_inc, cbox *inst_cbox)
{
int RetVal, NumIter, tmpIter, *IntPtr, IntCtr, IntRes;

   inout << "How many iterations?";
   inout >> NumIter;

   tmpIter = NumIter;
   while (tmpIter--)
      ((inc*)actor_inc)->incpriv(*(handle *)inst_cbox);

   tmpIter = NumIter;
   while (tmpIter--)
      ((inc*)actor_inc)->decpriv(*(handle *)inst_cbox);

   tmpIter = NumIter*2;
   IntCtr = 0;
   while (tmpIter--)
   {
      IntCtr ++;
      IntPtr = inst_cbox->rcvfromcbox();
      IntRes = *IntPtr;
      inout << "Return Value " << IntCtr << " is " << IntRes << "\n";
```

```
   }

   return 0;
}
/**********************************************************************
*
* fctrl --
*    computes the factorial for the number input by the user
**********************************************************************
/
app::fctrl(cbox *inst_cbox)
{
int number;

   inout << "What factorial would you like to compute? ";
   inout >> number;

   actor* actor_fact = New(fact());
   ((fact*)actor_fact)->compute_factorial(*(handle *)inst_cbox, number);
   int *IntPtr = inst_cbox->rcvfromcbox();
   int IntRes = *IntPtr;

   inout << "The factorial of " << number << " is " << IntRes << "\n";

}
/**********************************************************************
*
* done--
*    always when quitting, get the am handle and delete the application
**********************************************************************
/
app::done()
{
   inout << "APP(C) : Quitting Application - CIAO\n\n";

   app_mgt* am
     = (app_mgt*)(APPL_TABLE->get_family_handle(my_appl_id(),APP_MGR));
   am->delete_app(my_appl_id());
}

//Class FACT is derived from behavior

#ifndef FACT
#define FACT
class fact : public behavior
{
   local_streams inout;
public:
   fact();         // constructor
   ~fact();          // destructor
   compute_factorial(handle,int);
```

```
};
#endif

//*************************************************************
//definitions of methods of class fact
//*************************************************************
#include "utils.h"
#include "esk_base.h"
#include "msg_stuff.h"

#include "cbox.cc"
#include "behavior.cc"

#include "fact.h"
#include "prod.h"

//*************************************************************
//   constructor for fact behavior
//*************************************************************
fact::fact () : inout(this)
{
  alter_stack_size(8300);
  CONS_RETURN;
};

//******************************************************************
// destructor  for fact actor
//******************************************************************
fact::~fact() {};

//******************************************************************
// method that computes the factorial of a number passed to it
//******************************************************************
int fact::compute_factorial(handle inst_cbox, int number)
{
  actor *actor_prod = New (prod());
  ((prod *)actor_prod)->compute_product(inst_cbox,1,number);

  /*
    Doing a become before quitting
  */
  become (new {local} fact ());

  INT_RETURN(1);
}

//Class FILO is derived from behavior
#include "behavior.h"

#ifndef FILO
#define FILO
```

```
class filo : public behavior
{
  local_streams  inout;
  int     fil_id; /* unique identifier for the philosopher */
  handle    r_f;    /* right fork actor */
  handle    l_f;    /* left fork actor  */
  int     msg_tfc;
public:
  filo(int, handle, handle, int); // constructor for filo actor
  ~filo();          // destructor for filo actor
  int think_eat(handle);
};
#endif

//*****************************************************************
// Definitions of methods of class filo
//*****************************************************************
#include "utils.h"
#include "esk_base.h"
#include "msg_stuff.h"
#include <sys/time.h>

#include "cbox.cc"
#include "behavior.cc"

#include "filo.h"
#include "fork.h"

extern int gettimeofday (timeval*,timezone*);

//*****************************************************************
//   constructor for filo behavior
//*****************************************************************
filo::filo(int PhilId, handle RtFork, handle LtFork, int MsgTfc) :
inout(this)
{
  alter_stack_size(8300);

  fil_id = PhilId;
  r_f = RtFork;
  l_f = LtFork;
  msg_tfc = MsgTfc;

  CONS_RETURN;
};

//*****************************************************************
// destructor  for filo actor
//*****************************************************************
filo::~filo() {};
```

```
//**********************************************************************
// think and eat process of the philosopher
//**********************************************************************
int filo::think_eat(handle repbox)
{
timeval mytime;
unsigned int msectime, *IntPtr;
cbox *filo_cbox;
int RetCode;

   filo_cbox = (cbox *) new {remote} cbox();

   /*
     Wait For A Random Amount Of Time
   */
   gettimeofday (&mytime, NULL);
   msectime = (mytime.tv_sec % 10000) * fil_id;
   //inout << "msectime is " << msectime << " \n";
   while (--msectime);

   /*
     Attempt to lift forks to start eating
   */
   ((fork *)&r_f)->lift(*(handle *)filo_cbox);
   ((fork *)&l_f)->lift(*(handle *)filo_cbox);

   IntPtr = filo_cbox->rcvfromcbox();
   IntPtr = filo_cbox->rcvfromcbox();

   /*
     The philosopher has commenced eating - announce it
   */
   inout << "Philosopher " << fil_id << " is now EATING\n";

   /*
     Eat For A Random Amount Of Time
   */
   gettimeofday (&mytime, NULL);
   msectime = (mytime.tv_sec % 10000) * (fil_id + 2);
   //inout << "msectime is " << msectime << " \n";
   while (--msectime);

   /*
     Done eating - release forks now
   */
   ((fork *)&r_f)->release(*(handle *)filo_cbox);
   ((fork *)&l_f)->release(*(handle *)filo_cbox);

   IntPtr = filo_cbox->rcvfromcbox();
   IntPtr = filo_cbox->rcvfromcbox();
```

```
   /*
      The philosopher has started thinking - announce it
   */
   inout << "Philosopher " << fil_id << " is now THINKING\n";

   /*
      if we are done, respond to the cbox else send a message to myself
      and do a become
   */
   if (msg_tfc-1)
   {
      ((filo *)&get_parent())->think_eat(repbox);
      become (new {remote} filo(fil_id,r_f,l_f,--msg_tfc));
   }
   else
      RetCode = ((cbox *)&repbox)->reply2cbox(1);

   INT_RETURN(1);

}

// Class FORK is derived from behavior
#include "behavior.h"

#ifndef FORK
#define FORK
class fork : public behavior
{
   local_streams  inout;
   int            fork_no;
   int            used;
public:
   fork(int,int);       // constructor for fork actor
   ~fork();           // destructor for fork actor
   lift(handle);           // cbox handle to reply to
   release(handle);        // cbox handle to reply to
};
#endif

//*****************************************************************
// Definitions of methods of class fork
//*****************************************************************
#include "utils.h"
#include "esk_base.h"
#include "msg_stuff.h"

#include "cbox.cc"
#include "behavior.cc"

#include "fork.h"
#include "filo.h"
```

```
//******************************************************************
//   constructor for fork behavior
//******************************************************************
fork::fork(int ForkNum, int Used) : inout(this)
{
   alter_stack_size(8300);
   fork_no = ForkNum;
   used = Used;

   if (used)
   ((behset *)&my_behset)->bset_lock_method(METH_ID((int)&fork::lift));
   else
   ((behset *)&my_behset)-
>bset_lock_method(METH_ID((int)&fork::release));

   CONS_RETURN;
};

//******************************************************************
// destructor  for fork actor
//******************************************************************
fork::~fork() {};

//******************************************************************
// simulate the fork lift and become an unliftable fork
//******************************************************************
int fork::lift(handle inst_cbox)
{
int RetCode;

   RetCode = ((cbox *)&inst_cbox)->reply2cbox(1);
   /*
     become a fork which is ok for releasing only
   */
   become (new {local} fork(fork_no, 1));

   INT_RETURN(1);
}
//******************************************************************
// simulate the fork release and become a liftable fork
//******************************************************************
int fork::release(handle inst_cbox)
{
int RetCode;

   RetCode = ((cbox *)&inst_cbox)->reply2cbox(1);
   /*
     become a fork which is ok for lifting only
   */
   become (new {local} fork(fork_no, 0));
```

```
  INT_RETURN(1);
}

//Class PROD is derived from behavior

#ifndef PROD
#define PROD
class prod : public behavior
{
  local_streams inout;
public:
  prod();               // constructor
  ~prod();               // destructor
  compute_product (handle,int,int);
};
#endif

//*******************************************************************
//Definitions of methods of class prod
//*******************************************************************
#include "utils.h"
#include "esk_base.h"
#include "msg_stuff.h"

#include "cbox.cc"
#include "behavior.cc"

#include "prod.h"

//*******************************************************************
//   constructor for prod behavior
//*******************************************************************
prod::prod () : inout(this)
{
  alter_stack_size(8300);
  CONS_RETURN;
};

//*******************************************************************
// destructor  for prod behavior
//*******************************************************************
prod::~prod() {};

//*******************************************************************
// method that computes the product of numbers in the range low to high
//*******************************************************************
int prod::compute_product(handle inst_cbox, int low, int high)
{
int RetCode, mid, sub1, sub2, *ResPtr;
actor *actor_prod1;
```

```
actor *actor_prod2;
cbox  *sub_cbox;

  if (low >= high)
    RetCode = ((cbox *)&inst_cbox)->reply2cbox(low);
  else
  {
    mid = (low+high)/2;

    actor_prod1 = New (prod());
    actor_prod2 = New (prod());

    sub_cbox = (cbox *) new {local} cbox();

    ((prod *)actor_prod1)->compute_product
           (*(handle *)sub_cbox,low,mid);
    ((prod *)actor_prod2)->compute_product
           (*(handle *)sub_cbox,mid+1,high);

    ResPtr =  sub_cbox->rcvfromcbox();
    sub1 = *ResPtr;
    ResPtr =  sub_cbox->rcvfromcbox();
    sub2 = *ResPtr;

    RetCode = ((cbox *)&inst_cbox)->reply2cbox(sub1 * sub2);
  }

  INT_RETURN(1);
}
```

VITA

Nandan Joshi was born on 20th December, 1964 in Bombay, India. He completed his Bachelor's in Mechanical Engineering from the Victoria Jubilee Technical Institute in Bombay. He came to Virginia Tech in Fall 1986 to pursue a Master's degree in Mechanical Engineering. After completing his Master's degree in April 1988, he began working on his Master's degree in Computer Science. For the past three years, he has been working as a Software Engineer in Blacksburg, Virginia.