Technical Report 844

Actors: A Model Of Concurrent Computation In Distributed Systems

Gul A. Agha

MIT Artificial Intelligence Laboratory

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 844	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ACTORS: A MODEL OF CONCURRENT COMPUTATION IN DISTRIBUTED SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Gul Abdulnabi Agha		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0505
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE June 1985
		13. NUMBER OF PAGES 198
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Distributed Systems	Object-oriented Programming
Concurrency	Deadlock
Programming Languages	Semantics of Programs
Processor Architecture	Functional Programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A foundational model of concurrency is developed in this thesis. We examine issues in the design of parallel systems and show why the actor model is suitable for exploiting large-scale parallelism. Concurrency in actors is constrained only by the availability of hardware resources and by the logical dependence inherent in the computation. Unlike dataflow and functional programming, however, actors are dynamically reconfigurable and can model shared resources with changing local state. Concurrency is

20. spawned in actors using asynchronous message-passing, pipelining, and the dynamic creation of actors.

We define an abstract actor machine and provide a minimal programming language for it. A more expressive language, which includes higher level constructs such as delayed and eager evaluation, can be defined in terms of the primitives. Examples are given to illustrate the ease with which concurrent data and control structures can be programmed.

To send a communication, an actor must specify the target. Communications are buffered by the mail system and are eventually delivered. Two different transition relations are needed to model the evolution of actor systems. The possibility transition models events from some view-point. It captures the nondeterminism in the order of delivery of communications. The Subsequent transition captures fairness arising from the guarantee of delivery. We provide a denotational semantics for our minimal actor language in terms of the transition relations.

Abstraction in actors is achieved by a model in which the only observable communications are those between actors within a system and actors outside it. Our model makes no closed-world assumption since communications may be received from the outside at any point in time. The model provides for the composition of independent modules using message-passing between actors that interface the systems composed with their external environment.

This thesis deals with some central issues in distributed computing. Specifically, problems of divergence and deadlock are addressed. For example, actors permit dynamic deadlock detection and removal. The problem of divergence is contained because independent transactions can execute concurrently and potentially infinite processes are nevertheless available for interaction.

Actors : A Model Of Concurrent Computation In Distributed Systems

Gul A. Agha

This dissertation was submitted to the University of Michigan in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computer and Communication Science.

The report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's aritificial intelligence research is provided in part by the the System Development Foundation and in part by the Advanced Research Projects Agency of the Department of Defence under Office of Naval Research contract N0014-80-C-0505.

To All Sentient Life

In Memory Of

Sachal

(Aug 1983 - May 1984)

PREFACE

It is generally believed that the next generation of computers will involve massively parallel architectures. This thesis studies one of the proposed paradigms for exploiting parallelism, namely the actor model of concurrent computation. It is our contention that the actor model provides a general framework in which computation in distributed parallel systems can be exploited. The scope of this thesis is limited to theoretical aspects of the model as opposed to any implementation or application issues.

Many observers have noted the computational power that is likely to become available with the advent of a new generation of computers. This work makes a small contribution in the direction of realizing technology which seems just on the horizon. The possibilities that emerge from the availability of a massive increase in computational power are simply mind boggling. Unfortunately, humankind has generally lacked the foresight to use the resources that science has provided in a manner that would be compatible with its long-term survival. Somehow we have to develop an ethic that values compassion rather than consumption, to acquire a reverence for life itself. Otherwise this work, among others, will be another small step in the global march towards self-destruction.

The research reported in this thesis was carried out for the most part at M.I.T., where I have been working with the *Message-Passing Semantics Group*. The group is currently implementing the *Apiary architecture* for *Open Systems*, which is based on the actor model. Much of the development of the actor paradigm has been inspired by the work of Carl Hewitt whose encouragement and constructive criticism has been indispensable to the development of the ideas in this thesis. Carl Hewitt also read and commented on drafts of this thesis.

This thesis has been influenced by other work in the area of concurrency, most notably that of Robin Milner. Although we have shied away from using a λ-calculus like notation for an actor calculus, the transition system we develop has a similar flavor. Our preference has been for using a

programming language notation for purposes of overall clarity in expressing simple programs.

John Holland has provided both intellectual impetus and moral support over the years; in particular, numerous useful discussions with John have led to a better perspective on ideas in the field. I am also indebted to William Rounds for numerous suggestions, among them to develop a simple actor language and to illustrate its flavor by treating a number of commonly understood examples. My first thorough exposure to object-oriented architectures was in a course offered by Paul Scott. Conversations with Robin Milner, Vaughn Pratt, and Joe Stoy have provided critical feedback. Will Clinger's thesis interested me in the area of actor semantics. Members of the Message-Passing Semantics Group at M.I.T. have created an atmosphere which made the work described here possible. In particular, Henry Lieberman, Carl Manning, Chunka Mui and Thomas Reinhardt provided helpful comments.

The work described in here was made possible by generous funding from the System Development Foundation and by the support of the Artificial Intelligence Laboratory at M.I.T.

Finally, the time during which the ideas in this thesis were developed was a rather intense time in the lives of my family. Nothing would have been possible without the patient cooperation of my wonderful wife Jennifer Cole. It must be added that it was only due to the high spirits maintained by our son Sachal through most of his short, difficult life that any work at all could have been done by me.

Gul Agha
Cambridge, Massachusetts
March 1985.

Contents

Preface iii

Table Of Contents v

List Of Figures viii

1 Introduction 1

2 General Design Decisions 8

2.1 The Nature of Computing Elements 9

2.1.1 Sequential Processes 9

2.1.2 Functions Transforming Data Values 10

2.1.3 Actors 12

2.2 Global Synchrony and Asynchrony 14

2.3 Interaction Between Agents 17

2.3.1 Shared Variables 18

2.3.2 Communication 18

2.3.3 The Need for Buffering 20

2.4 Nondeterminism and Fairness 22

2.4.1 The Guarantee of Delivery 23

2.4.2 Fairness and the Mail System 24

2.5 Reconfigurability and Extensibility 26

2.5.1 A Resource Manager 26

2.5.2 The Dynamic Allocation of Resources 29

3 Computation In Actor Systems 32

3.1 Defining an Actor System 33

3.1.1 Tasks 33
3.1.2 The Behavior of an Actor 36
3.2 Programming With Actors 43
3.2.1 The Basic Constructs 44
3.2.2 Examples 51
3.3 Minimal Actor Languages 58
3.3.1 A Simple Actor Language 59
3.3.2 Act 63

4 A More Expressive Language **66**
4.1 Several Incoming Communications 67
4.1.1 A Static Topology 67
4.1.2 A Dynamic Topology 70
4.2 Insensitive Actors 75
4.3 Sequential Composition 81
4.4 Delayed and Eager Evaluation 84
4.4.1 Primitive Actors 85
4.4.2 Delayed Evaluation 87
4.4.3 Representing Infinite Structures 89
4.4.4 Eager Evaluation 95

5 A Model For Actor Systems **98**
5.1 Describing Actor Systems 100
5.1.1 Configurations 100
5.1.2 Requirements for a Transition Relation 102
5.2 Initial Configurations 103
5.2.1 Formalizing Actor Behaviors 104
5.2.2 The Meaning of Behavior Definitions 108
5.2.3 Mapping Actor Programs 115
5.3 Transitions Between Configurations 117
5.3.1 Possible Transitions 119
5.3.2 Subsequent Transitions 122

6 Concurrency Issues **127**
6.1 Problems in Distributed Computing 128
6.1.1 Divergence 128
6.1.2 Deadlock 132

6.1.3 Mutual Exclusion 137
6.2 Graphic Representations 138
6.2.1 Streams 138
6.2.2 Message Channels 140

7 Abstraction And Compositionality **144**
7.1 Abstraction 145
7.1.1 Atomicity 146
7.1.2 Nesting Transactions 148
7.2 Compositionality 150
7.2.1 Actors and Ports 151
7.2.2 Encapsulation in Actors 152
7.2.3 Composition Using Message-Passing 154
7.2.4 Rules for Composition 155
7.3 The Brock-Ackerman Anomaly 161
7.4 Observation Equivalence 166

8 Conclusions **172**

A Asynchronous Communication Trees **176**

References **187**

List of Figures

2.1 *An indeterminate applicative program* 12
2.2 *History sensitive functions* 13
2.3 *A synchronizing mechanism* 16
2.4 *A static graph linking the resource-manager to two devices.* 28

3.1 *An abstract representation of an actor* 38
3.2 *An abstract representation of transition.* 39
3.3 *Actor event diagrams* 42
3.4 *A factorial computation* 56

4.1 *A fixed topology for a two input function.* 69
4.2 *Dynamically accepted input* 71
4.3 *Implementing call expressions* 73
4.4 *Insensitive actors* 78
4.5 *Eager evaluation* 96

6.1 *A perpetual loop.* 130
6.2 *Dining Philosophers.* 133
6.3 *Streams and replacement in actors* 140
6.4 *Communication channels* 142

7.1 *Synchronous composition* 150
7.2 *Encapsulation in actor systems* 153
7.3 *The Brock-Ackerman anomaly.* 165

A.1 *A typical Asynchronous Communication Tree* 178
A.2 *Possible nondeterministic transitions.* 182
A.3 *A new receptionist definition.* 183
A.4 *Acceptance of a communication by an external actor.* 183

Chapter 1

Introduction

The purpose of any language is to communicate; that of a programming language is to communicate to a computer actions it ought to perform. There are two different sorts of objectives one can emphasize in the design of a programming language: *efficiency* in execution, and *expressiveness.* By "efficiency," we refer here only to the speed with which the actions implied in a program can be carried out by the computer. In a precise sense, the most efficient programming language would be one that literally told the computer what actions to carry out; in other words, a machine language.[1] Expressiveness refers to the ease with which an program can be understood and shown to behave correctly. A programming language is expressive to the extent that it can be used to specify reasonable behaviors in the simplest possible terms.

[1]Of course, every kind of processor has its own machine language. Some of these languages may be "inherently" more efficient than others.

A programming language that maximized efficiency would not necessarily lead to the specification of programs with the best *performance*. This is simply because the programmer may end up spending more time figuring out *how* to express rather than *what* to express. The best gains in performance are to be achieved by discovering less computationally complex methods of achieving the same result.

By and large, the goal of introducing new programming languages has been to make it simpler to express more complex behavior. Historically, the class of actions computers were first expected to carry out was that of computing well-defined mathematical functions. However, such computations are no longer the only tasks a modern computer performs. In fact, the storage of information, sorting and searching through such information, and even exploration of an imprecisely defined domain in real-time are emerging as significant applications. For example, computerized *databases*, such as the records maintained by a state Motor Vehicle Bureau, and *artificial intelligence* applications, such as computerized vehicles pioneering the navigation of Martian surface, are common uses of the computer. This more general use of computer programs has, in and of itself, important consequences for the class of behaviors we are interested in expressing.

Although newer programming languages have generally favored considerations of expressiveness over those of efficiency, the ability to solve complex problems by means of the computer has nevertheless increased. This remarkable trend has been achieved by creating faster and bigger processors. However, there is now good reason to believe that we may have approached the point of diminishing returns in terms of the size and speed of the individual processor. Already, smaller processors would be far more cost-effective, if we could use large numbers of them cooperatively. In particular, this implies being able to use them in parallel.

This brings us to the central topic of consideration in this thesis; namely, the development of a suitable language for *concurrency*. By concurrency we mean the *potentially* parallel execution of desired actions. Actually, concurrency by itself is not the real issue; after all concurrency has been exploited for a long time in the software revolution caused by time-sharing. The key difference between the now classic problem of operating systems, and our desire to exploit concurrency, is that in the former there is little interaction between the various "jobs" or "processes" that are executed concurrently. Indeed, the correctness of an operating system is dependent on making sure that none of the numerous (user-defined) processes affect each other.

Our problem is quite the reverse: we wish to have a number of processes work together in a meaningful manner. This doesn't really imply that there are no important lessons to be learned from operating system theory. For example, notice that we switched from talking about "processors" to talking in terms of "processes". A *processor* is a physical machine while a *process* is an abstract computation. From operating systems, we know that we may improve over-all performance of a processor by executing several processes concurrently instead of sequentially. How the processors are utilized is an issue for the underlying *network architecture* supporting the language. Our interest is in a model of concurrency that exploits concurrently executed processes without assuming anything about their concrete realization. The processes may be distributed over a network of processors which can be used in parallel; however, if our programming language did *not* support concurrency, such a distributed architecture would not result in any improvement in performance over a single processor.

Actually, we are not so much concerned with a particular programming language, but rather, with the meta-linguistic issues behind the constructs of a concurrent language. The operational semantics of a language defines an instruction set for computation on an *abstract machine*. (More precisely, in case of the actor model, a system of machines). We are interested in the characteristics of the underlying models of computation. Specifically, we will examine the issues of expressiveness and efficiency in the context of concurrent computation.

There are some intrinsic reasons for a theory of concurrency as well. One of these is the relevance of concurrency to an understanding of intelligent systems and communities. In particular, natural systems that appear to learn or adapt are all intrinsically parallel, and in fact quite massively so: the brain of animals, ecological communities, social organizations whether these are of human or non-human animals, are all examples of distributed systems that exploit concurrency. In fact, the *genetic algorithm* which is the foundation for adaptation and natural selection is itself intrinsically parallel [Holland 75]. The success of these mechanisms is sufficient grounds to interest one in the study of the implications of concurrent processing.

The rest of this chapter gives an overview of the thesis. The next chapter reviews the general design decisions that must be made in any model of concurrent computation. In Chapter 3, we describe the behavior of an actor and define a simple actor language which is used to show some specific examples of actors. In the following chapter, we then define several higher level constructs which make the actor language more expressive, and provide a mechanism for abstraction in actor systems. These constructs are definable in terms of the primitive actor constructs and are not considered as part of the *actor formalism.* Chapter 4 also defines an expressional language, and discusses different strategies for the evaluation of expressions.

Chapter 5 defines an *operational semantics* for actors by specifying a transition relation on configurations of actor systems. The guarantee of mail delivery is formalized by defining a second transition system which expresses this property. We take the primitive constructs of an actor language and show how one can provide these an with operational definition.

In chapter 6, we are concerned with issues raised in related models. There are some significant difficulties in exploiting concurrency: Distributed systems often exhibit pathological behavior such as divergence and deadlock. The actor model addresses these problems at a variety of levels. Divergence can be a useful property because of the guarantee of delivery; deadlock in a strict sense does not exist in an actor system. Besides, the asynchronous, buffered nature of communication in actors provides mechanisms to detect deadlock in a semantic sense of the term. Chapter 6 also explores the relation between some aspects of dataflow and actors; in particular, the similarity between replacement in actors and what has been claimed to be the "side-effect free" nature of computation in both systems.

Chapter 7 tackles the issue of abstraction and compositionality in actor systems. In particular, we discuss the nature of *open systems* and relate it to the insufficiency of the *history relation* observed in [Brock and Ackerman 77]. The right level of abstraction would permit us to treat equivalent systems as semantically identical and yet differentiate between systems that are unequal. We discuss the nature of composition in actors and show how we can model composition based on message-passing.

The final chapter summarizes some of the implications of the work in this thesis. The Appendix uses tools from Milner's work to define an abstract representation for actor systems in terms of what are called *Asynchronous Communication Trees.* This representation provides a suitable way of visualizing computations in actors.

Contributions

The specific contributions of this thesis are summarized below. This thesis provides:

- A critical overview of the various proposed models of concurrency.
- A simple outline of the actor model and the specification of minimal primitive constructs for actor languages.
- A transition system for actor systems and a structured operational semantics for an actor language.

- A paradigm for addressing problems in distributed computing which is suitable for computation in open systems.
- A model to support compositionality and abstraction from irrelevant detail.

Chapter 2

General Design Decisions

Several radically different models of concurrent computation have been proposed. In this chapter, we will review the concepts underlying each of the proposed models. Our interest is in comparing and contrasting their primitives with a view towards determining their generality. Of particular concern to us is the relative ease with which massively parallel architectures can be exploited. The design decisions fundamental to any model of concurrent computation include:

- the nature of the computing elements
- global synchrony versus asynchronous elements
- the mode of interaction between computing elements
- degree of fairness
- reconfigurability and extensibility

This list is by no means exhaustive but represents the aspects we think are the most significant. There are other issues, such as the linguistic issues in the specification of a language based on each of the models, but we will ignore such details in our present discussion. We discuss each of the design issues in the sections that follow.

2.1 The Nature of Computing Elements

The elements performing computations are, in an abstract *denotational* sense, some kind of a function. However, the domain and range of the functions defining the behavior of the elements is quite different in each of the models. Ignoring some significant details, we identify three distinct kinds of computational elements:

1. Sequential Processes.

2. Functions transforming data values.

3. Actors.

2.1.1 Sequential Processes

The *operational* notion of a *sequential process* is that it performs a sequence of transformations on *states*, where a state is a map from *locations* to *values* such as integers. In addition, the transformations may depend on certain "inputs" and produce "outputs." It is this latter aspect which makes the denotational semantics of systems of sequential process more difficult;

in particular, explicit consideration of the possibility of deadlock (when a process is waiting for input that never arrives) is required [Brookes 83]. Sequential processes are themselves, predictably, sequential in nature, but can execute in parallel with each other.

In a sense, sequential processes are inspired by algol-like procedures in sequential programming. Examples of systems based on the concept of sequential processes include *Concurrent Pascal* [Brinch Hansen 77], *Communicating Sequential Processes* [Hoare 77], and the *Shared Variables* model [Lynch and Fischer 81].

2.1.2 Functions Transforming Data Values

A second kind of computational element is a function which acts directly on data, without the benefit, or burden, of a store. Such functional models are derived from the λ-calculus based languages such as Pure Lisp [McCarthy 59]. Examples of concurrent systems using some variant of the functional model include *dataflow* [Agerwala and Arvind 82] and *networks of parallel processes* [Kahn and MacQueen 77]. In dataflow architectures, a *stream* of (data) values pass through functional agents [Weng 75]. The concurrency in the system is a result of being able to evaluate the arguments to the functions in parallel.

Perhaps the simplest model of systems using functions is an *indeterminate applicative system* where the call-by-value is used to evaluate the arguments and the result of the computation is a single value. Computation in such systems *fans in* as arguments are evaluated and passed along.

Fig. 2.1 shows an example of concurrent evaluation in an *indeterminate applicative system.*

The functional elements may take several parameters as inputs but, given the parameters, can output only a single value. The same value may, however, be sent to different computational elements. Unfortunately, functions are *history insensitive* [Backus 78]. This can be a problem when modeling the behavior of systems that can change their behavior over time. For example, consider the behavior of a *turnstile* with a counter which records the number of people passing through it. Each time the turnstile is turned, it reports a new number on the counter. Thus its behavior is not simply a function of a "turn" message but sensitive to the prior history of the computation. The *turnstile* problem is essentially equivalent to that of generating the list of all integers, producing them one at a time in response to each message received.

This problem is dealt with in some functional systems by feedback, using cyclic structures, as shown in Fig. 2.2 adapted from [Henderson 80]. The turnstile is represented as a function of two inputs, a "turn" message and an integer n. Its behavior is to produce the integer $n+1$ in response. The links act as *(first-in first-out) channels*, buffering the next value transmitted until the function has been evaluated and accepts more input. (The same value is sent down all the links at a fork in the diagram.)

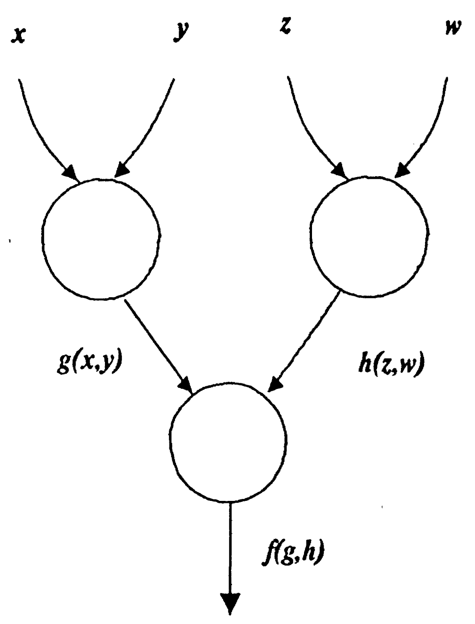

Figure 2.1: *An indeterminate applicative program. The parameters of the function are evaluated concurrently.*

2.1.3 Actors

Actors are computational agents which map each incoming communication to a 3-tuple consisting of:

1. a finite set of communications sent to other actors;
2. a new behavior (which will govern the response to the next communication processed); and,
3. a finite set of new actors created.

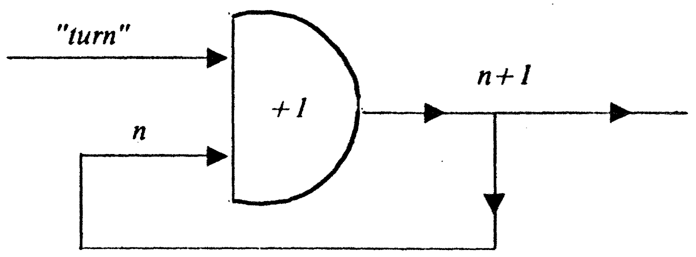

Figure 2.2: *History sensitive behavior as evaluation of a function with feedback.*

Several observations are in order here. Firstly, the behavior of an actor can be *history sensitive*. Secondly, there is *no presumed sequentiality* in the actions an actor performs since, mathematically, each of its actions is a function of the actor's behavior and the incoming communication. And finally, *actor creation* is part of the computational model and not apart from it. An early precursor to the development of actors is the concept of objects in SIMULA [Dahl, et al 70] which represented containment of data with the operations and procedures on such data in a single object.

Actors are a more powerful computational agent than sequential processes or value-transforming functional systems. In other words, it is possible to define a purely functional system as an actor system, and it is possible to specify arbitrary sequential processes by a suitable actor system, but it is not possible to represent an arbitrary actor system as a system of sequential processes or as a system of value-transforming functions. To see how actors can be used to represent sequential processes or functional programs is not difficult: both are special cases of the more general actor model. If the reader is not convinced of this, the machinery developed later in this thesis should make it clear.

It is easy to see why the converse is true: actors may create other actors; value-transforming functions, such as the ones used in *dataflow* can not create other functions and sequential processes, as in *Communicating Sequential Processes*, do not create other sequential processes.[1] In the sequential paradigm of computation, this fact would not be relevant because the same computation could be represented, mathematically, in a system without actor creation. But in the context of parallel systems, the degree to which a computation can be *distributed* over its lifetime is an important consideration. Creation of new actors guarantees the ability to abstractly increase the distributivity of the computation as it evolves.

2.2 Global Synchrony and Asynchrony

The concept of a unique global clock is not meaningful in the context of a distributed system of self-contained parallel agents. This intuition was first axiomatized in [Hewitt and Baker 77] and shown to be consistent with other *laws of parallel processing* in [Clinger 81]. The reasoning here is analogous

[1]Sequential processes may activate other sequential processes and multiple activations are permitted but the topology of the individual process is still static. The difference between activation and creation is significant in the extent of reconfigurability afforded by each.

to that in special relativity: information in each computational agent is localized within that agent and must be communicated before it is known to any other agent. As long as one assumes that there are limits as to how fast information may travel from one computational agent to another, the local states of one agent as recorded by another relative to its own local states will be different from the observations done the other way round.

We may conclude that, for a distributed system, a *unique (linear) global time* is not definable. Instead, each computational agent has a local time which linearly orders the events as they occur at that agent, or alternately, orders the local states of that agent. These local orderings of events are related to each other by the *activation ordering*. The activation ordering represents the causal relationships between events happening at different agents. Thus the global ordering of events is a partial order in which events occurring at different computational agents are unordered unless they are connected, directly or indirectly, because of one or more causal links.

This is not to imply that it is impossible to construct a distributed system whose behavior is such that the elements of the system can be abstractly construed as acting synchronously. An example of such a system is Cook's *hardware modification machine* [Cook 81]. The hardware modification machine is a mathematical abstraction useful for studying the problems of computational complexity in the context of parallelism.

The problem of constructing a synchronously functioning system is essentially one of defining protocols to cope with the fundamental epistemological limitation in a distributed system. To see how the elements of a system can be construed to be *synchronous*, consider the example shown in Fig. 2.3.

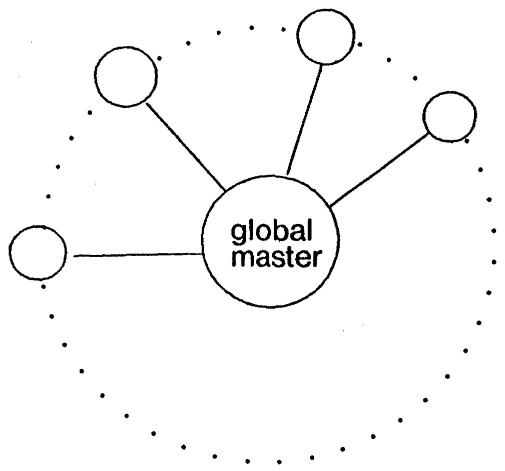

Figure 2.3: *A synchronizing mechanism: A Global Master controls the elements of the system.*

Assume one element, called the *global master*, controls when each of the elements in the system may continue; all elements perform some predetermined number of actions, report to the global master and wait for another "go" message from the global master before proceeding. The global master knows how many elements there are in the system and waits for each of them to report before sending out the next "go" message. Conceptually, we can think of each of the elements acting synchronously and the system passing through execution cycles on a "global clock". We can ignore the precise *arrival order* of messages to the global master, because in such a system the exact order may be irrelevant.

The important point to be made is that any such global synchronization creates a bottleneck which can be extremely inefficient in the context of a distributed environment. Every process must wait for the slowest process to complete its cycle, regardless of whether there is any logical dependence of a process on the results of another. Furthermore, it is not altogether obvious that such global synchrony makes it any easier to write programs in general. Although systems designed to act synchronously may be useful in some particular applications, we will deal with the general asynchronous distributed environment; the behavior of the synchronous system can always be derived as a special case. (See, for example, the discussion in chapter 4 of mechanisms involving an effectively prioritized exchange of communications between two actors.)

2.3 Interaction Between Agents

How the elements of a concurrent system affect each other is one of the most salient features of any model of concurrent computation. The proposed modes of interaction between the computational elements of a system can be divided into two different classes:

1. variables common to different agents; and,

2. communication between independent agents.

We take up these two modes of interaction in turn.

2.3.1 Shared Variables

The basic idea behind the *shared variables* approach is that the various processes can read and write to variables common to more than one process. When one process reads a variable which has been changed by another, its subsequent behavior is modified. This sort of common variables approach is taken in [Lynch and Fischer 81].

The shared variables approach does not provide any mechanism for abstraction and information hiding. For instance, there must be pre-determined protocols so that one process can determine if another has written the results it needs into the relevant variables. Perhaps, even more critical is the fact that this approach does not provide any mechanism for protecting data against arbitrary and improper operations. An important software principle is to combine the procedural and declarative information into well-defined *objects* so that access to data is controlled and modularity is promoted in the system. This sort of *absolute containment* of information is also an important tool for *synchronizing* access to scarce resources and proving freedom from *deadlock.* In a shared variables model, the programmer has the burden of specifying the relevant details to achieve meaningful interaction.

2.3.2 Communication

Several models of concurrent computation use communication between independent computational agents. Communication provides a mechanism by which each agent retains the integrity of information within it. There are two possible assumptions about the nature of communication between independent computational elements; communication can be considered to be either:

- *Synchronous*, where the *sender* and the *receiver* of a communication are both ready to communicate; or,
- *Asynchronous*, where the *receiver* does not have to be ready to accept a communication when the *sender* sends it.

Hoare's *Communicating Sequential Processes* and Milner's *Calculus of Communicating Systems* assume synchronous communication while the *actor model* [Hewitt 77] and *dataflow architectures* [Ackerman 84] do not.

Let's examine each assumption and its implications. A concurrent computational environment is meaningful only in the context of a conceptually distributed system. Intuitively, there can be no action at a distance. This implies that before a sender can know that the receiver is "free" to accept a communication, it must send a communication to the receiver, and *vice-versa.* Thus one may conclude that any model of synchronous communication is built-on asynchronous communication.

However, the fact that synchronous communication must be defined in terms of asynchronous communication does not necessarily imply that asynchronous communication is itself the right level of abstraction for programming. In particular, an argument could be made that synchronous communication should be provided in any programming language for concurrent computation if it provides a means of writing programs without being concerned with detail which may be required in all computation. The question then becomes if synchrony in communication is helpful as a universal assumption for a programming language. We examine this issue below.

2.3.3 The Need for Buffering

Every communication is of some finite length and takes some finite time to transmit. During the time that one communication is being sent, some computational agent may try to send another communication to the agent receiving the first communication. Certainly, one would not want to interleave the arbitrary bits of one communication with those of another! In some sense, we wish to preserve the *atomicity* of the communications sent. A solution to this problem is to provide a "secretary" to each agent which in effect tells all other processes that the agent is "busy."[2] Essentially, the underlying system could provide such a "secretary" in an implementation of a model assuming synchronous communication, as in a telephone network.

There is another problem in assuming synchronous communication. Suppose the sender is transmitting information faster than the receiver can accept it. For example, as this thesis is typed in on a terminal, the speed of the typist may at times exceed the rate at which the computer is accepting the characters. To get around this problem, one could require that the typist type only as fast as the *editing process* can accept the characters. This solution is obviously untenable as it amounts to typing one

[2]This could be done for instance by simply not responding to an incoming communication.

character at a time and waiting for a response (in fact, the argument would continue to the level of electrons!). The other solution is to provide the system with the capability to buffer the segments of a communication.

Of course, if the underlying system is required to buffer segments of a communication, it can equally well be required to buffer different communications so that the sender does not *have* to be "busy waiting" for the receiver to accept a communication before it proceeds to do some other processing. Thus *buffered asynchronous communication* affords us efficiency in execution by *pipelining* the actions to be performed. Furthermore, synchronous communication can be defined in the framework of asynchronous communication.[3] The mechanism for doing so is simply "freezing" the sender until the receiver acknowledges the receipt of a communication [Hewitt and Atkinson 77].

There is yet another significant advantage in buffered asynchronous communication. It may be important for a computational element to communicate with itself; in particular, this is the case when an element defines a recursive computation. Communication with oneself is however impossible if the *receiver* must be free when the *sender* sends a communication: this situation leads, immediately, to a *deadlock* because the *sender* will be "busy waiting" forever for itself to be free. The problem actually is worse:

[3]The notion of synchrony as simultaneity is physically unrealizable. The *failure of simultaneity at a distance* occurs because whether two clocks are synchronous is itself dependent on the particular *frame of reference* in which the observations are carried out [Feynman, et al 1965]. We assume any notion of synchronous communication is a conceptual one.

no mutually recursive structure is possible because of the same reason. Mutual recursion, however, may not be so transparent from the code. There is no *a priori* problem with such recursive structures if the communications are buffered.

Both the *dataflow architecture* for functional programming [Ackerman 82] and the *apiary architecture* for actor systems [Hewitt 80] provide the capabilities to buffer communications from asynchronous computing elements. However, it is not altogether obvious how the computational elements to provide for buffering communications can be defined in a functional language (as opposed to simply assumed). Such buffers are readily defined in actor languages.

2.4 Nondeterminism and Fairness

Nondeterminism arises quite inevitably in a distributed environment. Conceptually, concurrent computation is meaningful only in the context of a distributed environment. In any real *network of computational agents*, one can not predict precisely when a communication sent by one agent will arrive at another. This is particularly true when the network is *dynamic* and the underlying architecture is free to improve performance by *reconfiguring* the virtual computational elements. Therefore, a realistic model must assume that the *arrival order* of communications sent is both arbitrary and entirely unknown. In particular, the use of the *arbiter* as the hardware element for serialization implies that the arrival order is physically indeterminate.

2.4.1 The Guarantee of Delivery

Given that a communication may be delayed for an arbitrarily long period of time, the question arises whether it is reasonable to assume that a communication sent is always delivered. In a purely physical context, the finiteness of the universe suggests that a communication sent ought to be delivered. However, the issue is whether buffering means that the *guarantee of delivery* of communications is impossible. There are, realistically, no *unbounded buffers* in the physically realizable universe. This is similar to the fact that there are no *unbounded stacks* in the universe, and certainly not in our processors, and yet we parse recursive control structures in algolic languages as though there were an infinite stack. The alternate to assuming unbounded space is that we have to assume some specific finite limit; but each finite limit leads to a different behavior. There is, however, no general limit on buffers: the size of any real buffer will be specific to any particular implementation and its limitations. The point of building a semantic model is to abstract away from such details inherent in any implementation.

The guarantee of delivery of communications is, by and large, a property of well-engineered systems that should be modeled because it has significant consequences. If a system did not eventually deliver a communication it was buffering, it would have to buffer the communication indefinitely. The cost of such storage is obviously undesirable. The guarantee of delivery does not assume that every communication is "meaningfully" processed.

For example, in the actor model, the processing of communications is dependent on the behavior of individual actors, and there may be classes of actors which ignore all communications or indefinitely buffer some communications. In particular, the guarantee of delivery provides one with mechanisms to reason about concurrent programs so that results analogous to those established by reasoning about the total correctness in sequential programs can be derived; in some cases, the guarantee helps prove termination properties.

2.4.2 Fairness and the Mail System

Not all algorithms for delivering communications result in a *mail system* that guarantees delivery. For instance, a mail system that always delivered a "shorter" communication in its buffer may not deliver all communications. Consider an agent, in such a system, which sent itself a "short" communication in response to a "short" communication. If a "long" and a "short" communication are concurrently sent to this actor, it may never receive the "long" communication.

The guarantee of delivery is one form of what is called *fairness*. There are many other forms of fairness, such as *fairness over arbitrary predicates*, or *extreme fairness* [Pnueli 83] where probabilistic considerations are used. The guarantee of delivery of communications is perhaps the weakest form of fairness one can define (although it is not clear to me what sort of formal framework one would define to establish this rigorously). The question arises if one should assume a stronger form of fairness; for example, that the communications sent are received in an probabilistically random order regardless of any property they have.

Consider a system that chooses to deliver up to three "short" communications for every "long" communication it delivers (if the shorter communications are found). Such a system would still satisfy the requirement of guaranteeing delivery of communications, but would not satisfy some stronger fairness requirements, for example, the requirement that all communications sent have an equal probability of being the next to be delivered. On the other hand, it may be very reasonable to have such an underlying *mail system* for some applications. We prefer to accept the guarantee of delivery of communications but not any form of fairness stronger than this guarantee. We will study the implications and usefulness of the guarantee later in this thesis.

Of course, given the lack of a unique order of events in a distributed system, what the definitions of stronger forms of fairness really mean is not altogether obvious. Our initial cognizance in such cases can sometimes be misleading because our intuitions are better developed for sequential processes whose behavior is qualitatively different. In particular, the mail system is itself distributed and the delivery of communications, even according to a given observer, may overlap in time.

2.5 Reconfigurability and Extensibility

The patterns of communication possible in any system of processes defines a topology on those processes. Each process (or computational agent) may, at any given point in its local time, communicate with some set of processes. As the computation proceeds, a process may either communicate only with the same processes it could communicate with at the beginning of the computation, or it may evolve to communicate with other processes that it could not communicate with before. In the former case, the *interconnection topology* is said to be *static*; and in the latter, it is *dynamic.*

Any system of processes is somewhat easier to analyze if its interconnection topology is static: the graph representing the connections between the processes is constant and hence relatively more information about the system is available at compile-time. Perhaps because of this structural simplicity in the analysis of static topologies, many models of concurrency assume that a process can communicate with only the same processes over its life-time. A static topology, however, has severe limitations in representing the behavior of real systems. We illustrate these limitations by means of the following example.

2.5.1 A Resource Manager

Consider the case of a *resource-manager* for two printing devices. We may assume for our present purposes that the two devices are identical in their behavior and therefore interchangeable. One would like this *resource-*

manager to

1. Send the *print requests* to the first available printing device.

2. When a *print request* has been processed, to send a *receipt* to the user requesting the printing.

These requirements imply that the *resource-manager* be able to communicate with a different device each time. Thus a system where the communication links were static and communications were sent down these links, without the *resource-manager* being able to choose which link ought to be used, would either send a communication to both the devices or to neither. This is the situation in a dataflow graph shown in Fig. 2.4. However, *resource-manager* should be able to choose where it wants to send a communication (depending on which device is free), suggesting that the *edges* represent only potential communication channels and not actual ones. The true links would be dynamically determined.

Suppose a system allowed the *resource-manager* to decide which of the two printing devices it wanted to communicate, with but relied on synchronous communication. The use of resources would be inefficient if the *resource-manager* was "busy waiting" for one particular printing device while the other one was idle. To get around this problem, suppose we required the *resource-manager* to keep a track of which device, if any, was idle and to attempt to communicate only with such a device. In this case, when a *busy* device becomes *idle*, it must inform the *resource-manager* that it is free. Once again, if the *resource-manager* is required to specify which

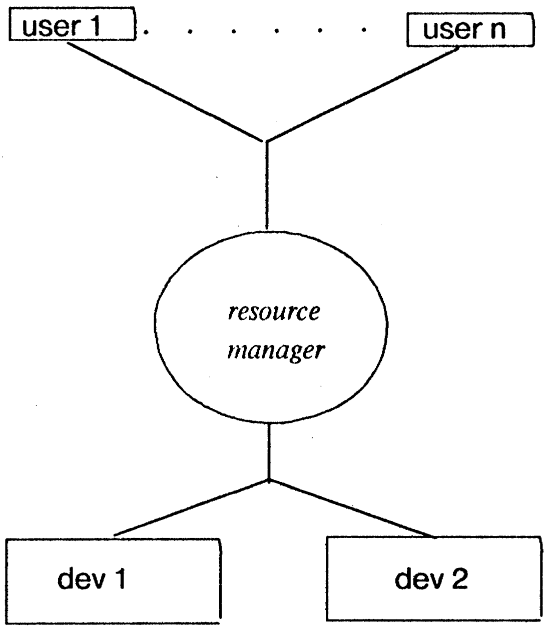

Figure 2.4: *A static graph linking the resource-manager to two devices.*

particular device it will accept input from, and be "busy waiting" to do so, the problem persists as it can not predict which one would be free first.

Requiring a receipt to the user introduces other complications. For one, the number of users will vary with time. This variation by itself creates the need for a dynamic graph on the processes [Brock 83]. For another, the maximum number of users need not be constant. In a system that might evolve to include more resources, the addition of the increased capacity should be graceful and not require the redefinition of the entire system. This implies that a solution using a fixed number of *communication* *channels* is not very satisfactory in an *open system* which is constantly subject to growth [Hewitt and de Jong 82]. For instance, if we wanted to add a third printing device, we should not necessarily have to program another *resource-manager* , but rather should be able to define a *resource-manager* which can incorporate the presence of a new printing device when sent an appropriate message to that effect.

A system that is not only *reconfigurable* but *extensible* is powerful enough to handle these problems. Reconfigurability is the logical prerequisite of extensibility in a system because the ability to gracefully extend a system is dependent on the ability to relate the extension to the elements of the system that are already in existence. An elegant solution to this problem of resource management using an actor system can be found in [Hewitt, et al 84].

2.5.2 The Dynamic Allocation of Resources

Extensibility has other important consequences. It allows a system to dynamically allocate resources to a problem by generating computational agents in response to the magnitude of a computation required to solve a problem. The precise magnitude of the problem need not be known in advance: more agents can be created as the computation proceeds and the maximal amount of concurrency can be exploited.

For example, consider a "balanced addition" problem, where the addition has to be performed on a set of real numbers. If the numbers are added sequentially,

$$(...(((a_1 + a_2) + a_3) + a_4) + ... + a_n)$$

then there is a classic problem of "propagation of errors," discussed in [von Neumann 58]. The problem occurs because real numbers are implemented using floating-point registers. Computational errors, instead of being statistically averaged, become fixed as rounding errors move to more significant bits. It is preferable to add the numbers in pairs,

$$(...(((a_1 + a_2) + (a_3 + a_4)) + ((a_5 + a_6) + (...))) \; + \; ... \; + \; (a_{n-1} + a_n)...)$$

which results in the error being statistically reduced by the "law of large numbers."

Addition in pairs is ideal for concurrent computation because it can be done using parallel computation in log-time, as opposed to linear time when done sequentially. Now if we had a program to carry out this addition in pairs, we may like the program to work even if we input a different number of real numbers each time. Thus we can not define a static network to deal with this problem [Emden and Filho 82]. Addition in pairs is easily accomplished in an actor system by creating other actors, called *customers*, and doing the evaluations concurrently. Such concurrency is the default in actor languages.

Reconfigurability in actor systems is obtained using the *mail system* abstraction. Each actor has a *mail address* which may be freely communicated to other actors, thus changing the interconnection network of the system of actors as it evolves. We will discuss the specific mechanisms later in this thesis.

Chapter 3

Computation In Actor Systems

In this chapter, we examine the structure of computation in the actor paradigm. The discussion here will be informal and intuitive, deferring consideration of the technical aspects to later chapters. The organization of this chapter is as follows. In first section, we explain actors and communications. The second section outlines the constructs which suffice to define a minimal actor language. We give some examples of actor programs to illustrate the constructs using only structured "pseudo-code." In the final section, kernels of two simple actor languages are defined and a program example is expressed in each of these languages. The two languages, SAL and *Act*, are both minimal yet are sufficient for defining all possible actor systems. SAL follows an algol-like syntax while *Act* uses a Lisp-like syntax. In the next chapter, we will define some new linguistic constructs, but these constructs will not be foundational; they can be defined using a minimal actor language. Such extensions to a minimal language demonstrate the power of the primitive actor constructs.

3.1 Defining an Actor System

Computation in a system of actors is in response to communications sent to the system. Communications are contained in *tasks.* As computation proceeds, an actor system evolves to include new tasks and new actors that are created as a result of processing tasks already in the system. All tasks that have already been processed (and all actors that are no longer "useful," a notion we will define more precisely), may be removed (i.e., *garbage collected*) from the system without affecting its subsequent behavior.[1] The *configuration* of an *actor system* is defined by the actors it contains as well as the set of unprocessed tasks.

3.1.1 Tasks

In somewhat simplified terms, we can say that the unprocessed tasks in a system of actors are the driving force behind computation in the system. We represent a *task* as a three tuple consisting of:

1. a *tag* which distinguishes it from all other tasks in the system;

[1]We refer here to the semantic equivalence of the systems with and without "garbage." Of course, the performance of the system is a different matter.

2. a *target* which is the mail address to which the communication is to be delivered; and,

3. a *communication* which contains information made available to the actor at the target, when that actor processes the given task.

As a simplification, we will consider a *communication* to be a tuple of values. The values may be mail addresses of actors, integers, strings, or whatever, and we may impose a suitable type discipline on such values. There are other possible models here; perhaps the most exciting of such models, and the one using the greatest uniformity of construction, is one in which the communications are themselves actors.[2] In such a model, communications may themselves be sent communications. For example, if we want a communication k_1 to print itself, we could send a communication k_2 to the communication k_1 which asked k_1 to print itself. Communications as actors also provide an effective and simple way to implement *call-by-need* using *futures*, where a future is a communication that can be sent a communication to evaluate itself. The semantic theory of actors is, however, considerably complicated by modelling communications as actors, and we therefore won't do so here.[3]

[2] The behavior of an actor is to send communications to other actors it knows about (i.e., its *acquaintances*), which in turn do the same until the communications are received by pre-defined *primitive actors* such as numbers and primitive pre-defined operations (See Section 4.4.). In the more general *universe of actors model*, tasks themselves are actors which have three *acquaintances*, namely the three components of the tuple given above.

[3] For a discussion of the *universe of actors* model see §4.4.

The *target* must be a valid mail address. In other words, before an actor can send the target a communication, it must know that the target is a valid mail address [Hewitt and Baker 77] . There are three ways in which an actor α, upon accepting a communication $\bar{k}$, can know of a target to which it can send a communication. These are:

- the target was known to the actor α before it accepted the communication $\bar{k}$,
- the target became known when α accepted the communication $\bar{k}$ because it was contained in the communication $\bar{k}$, or
- the target is the mail address of a new actor created as a result of accepting the communication $\bar{k}$.

A *tag* helps us to uniquely identify each task by distinguishing between tasks which may contain identical targets and communications. We will make use of the uniqueness of each tag when we define an operational semantics for actor systems. An important observation that should be made here is that any particular representation of the tags is somewhat arbitrary. The tags are specified because they are useful in keeping a track of tasks. However, the tasks themselves are existentially distinct entities.

There are various ways of representing tags; one such representation is a string of nonnegative integers separated by "." (periods). Using this representation, if ω is a tag for task t, then $\omega.n$, where n is some nonnegative integer, can be the tag for some task created as a result of processing t. In this way, if we start with a set of tags uniquely associated with the tasks, we can guarantee that all tasks always have distinct tags (by using a restriction that the last number appended is distinct for each task created by the same actor in response to the same communication). Note that there may be only a finite number of tasks in any given system.

3.1.2 The Behavior of an Actor

As we discussed earlier, all computation in an actor system is the result of processing communications. This is somewhat similar to a *data-driven* system like *dataflow*, and in contrast to systems based on processes that either terminate or are perpetually "active." Actors are said to *accept* a communication when they process a task containing that communication. An actor may process only those tasks whose target corresponds to its mail address. When an actor accepts a communication, it may create new actors or tasks; it must also compute a replacement behavior.

For any given actor, the *order of arrival* of communications sent to that actor is a linear order. In particular, this implies that the *mail system* must provide suitable mechanisms for *buffering* and *arbitration* of incoming communications when such communications arrive at roughly the same time. The *mail system* places the communications sent to a given target on the *mail queue* corresponding to that target. For most purposes, it is appropriate to consider the *mail queue* as part of the *mail system*. However, when we wish to deal with issues related to the *arrival order* of communications, such as the *guarantee of mail delivery*[4] we have to consider the mail queue

[4]The presence of communication failures in a real system should not be considered a hin-

explicitly.

An actor may be described by specifying:

- its mail address, to which there corresponds a sufficiently large *mail queue*[5]; and,
- its *behavior*, which is a function of the communication accepted.

Abstractly, we may picture an actor with a mail queue on which all communications are placed in the order in which they arrive and an *actor machine*[6] which points to a particular cell in the mail queue. The end of a communication on the mail queue can be indicated by some special symbol reserved for the purpose.[7] We represent this pictorially as in Fig. 3.1.

When an actor machine X_n accepts the nth communication in a mail queue, it will create a new actor machine, X_{n+1}, which will carry out the replacement behavior of the actor. This new actor machine will point to

drance for a theoretical investigation assuming a reliable mail system. See the discussion in Section 2.4.

[5]The mail queue will be considered large enough to hold all communications sent to a given actor. This implies that a mail queue is, in principle, unbounded, while only a finite fragment of it is used at any given point in time. This is quite similar to a read-only tape of a Turing Machine. However, the writing is done, indirectly, using the mail system.

[6]No assumption should be made about an actor machine being sequential, indeed, an actor machine, much like machines in the real world, may have components that function in parallel.

[7]Thus the variable length of a communication is not a problem.

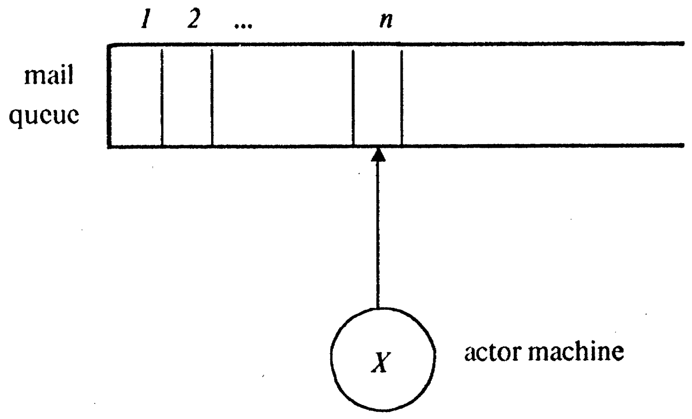

Figure 3.1: *An abstract representation of an actor. The actor machine contains information that determines the behavior of an actor. It accepts the current communication and can not process information from any other communication.*

the cell in the mail queue in which the *n+1st* communication is (or will be) placed. This can be pictorially represented as in Fig. 3.2.

The two actor machines X_n and X_{n+1} will not affect each others behavior: X_n processes only the nth communication. (Of course, if X_n sends the actor itself a communication, X_{n+1} may be the actor machine which processes the same.) Specifically, each of the actor machines may create their own tasks and actors as defined by their respective behaviors. Before the machine X_n creates X_{n+1}, X_n may of course have already created some actors and tasks; however, it is also the possible that X_n may still be in

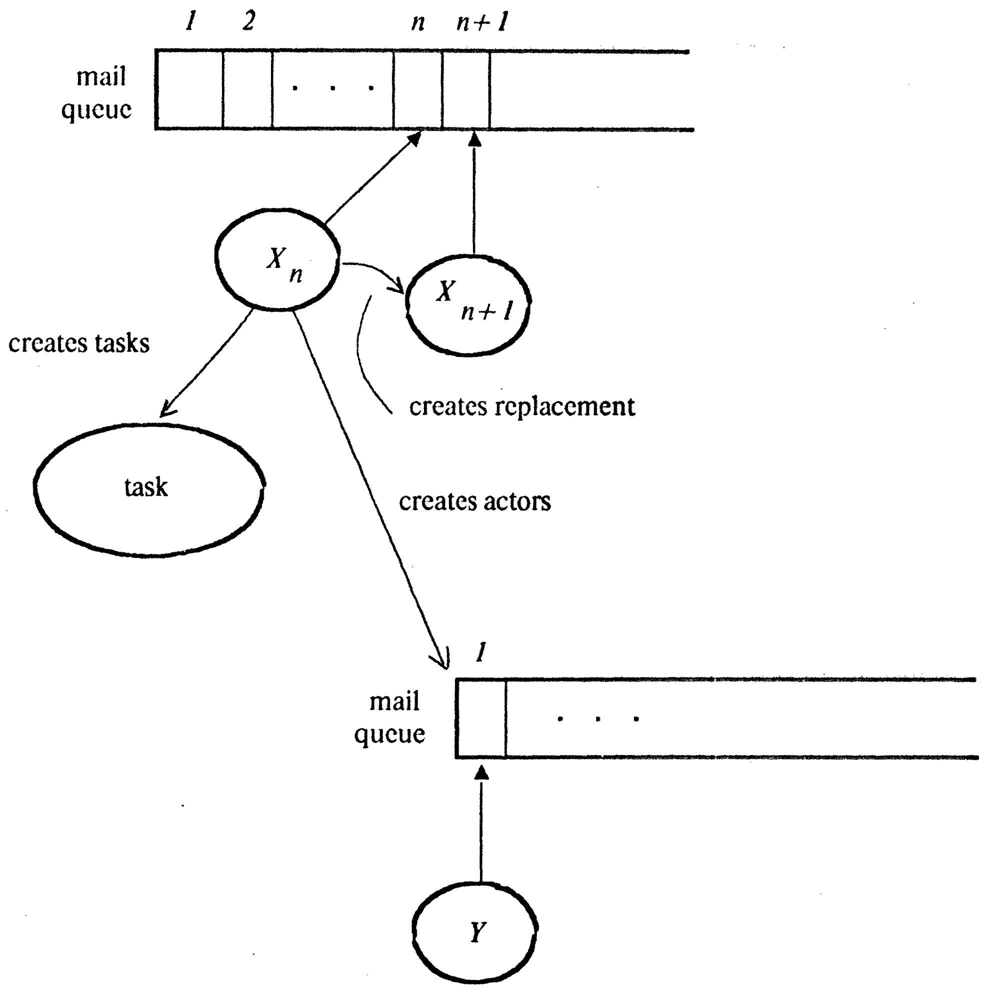

Figure 3.2: *An abstract representation of transition.*

the process of creating some more tasks and actors even as X_{n+1} is doing the same. In any event, note that the machine X_n will neither receive any further communications nor will it specify any other replacement.[8]

If we define an event as the creation of a new actor or task, or the specification of the replacement, then the order of events that are caused, at any actor, by the acceptance of communications is a partial order. The replacement machines at any mail address have a total order between them. This linear order is isomorphic to the arrival order of the corresponding communications which result in their replacement (as may be readily inferred from the Fig. 3.2).

An event-based picture for computation in actors uses *life-lines* which are shown in Fig. 3.3. Each actor has an order of acceptance of communications which is linear. The events in the life of an actor are recorded in the order in which they occur: the further down the line, the later in local time. Activations (causal ordering of events) are indicated by the lines connecting two different actors with the arrow on the line indicating causal direction. Finally, each lifeline is labeled by the pending communications, i.e., the communications that have been received but not processed. Clinger [81] used collections of life-lines to provide a fixed-point semantics for actors. The resulting pictures are called the *actor event diagrams*.

A couple of general remarks about the implementation issues are in order here:

Remark 1. The reader may wonder about the efficiency of constructing a new actor machine in response to each communication accepted. It should

[8]We will later model functions that require more input as a collection of these elemental actors.

be emphasized that this is simply a conceptual assumption that frees us from the details of any particular implementation. *Concurrency* simply means potential parallelism. Some implementations may find it useful to generally delay constructing the replacement until the old machine can be cannibalized. However, delaying the construction of the replacement is not a universal requirement as would be the case in a sequential machine. Thus, if there are sufficient resources available, computation in an actor system can be speeded up by an order of magnitude, by simply proceeding with the next communication as soon as the ontological necessity of determining the replacement behavior has been satisfied. The advantages of this kind of *pipelining* can be illustrated by the following simple example: Consider a calculation which requires $\mathcal{O}(n^2)$ sequential steps to carry out, where $\mathcal{O}(n)$ represents the size of input. Suppose further that computing the replacements takes only $\mathcal{O}(n)$ steps. If we had a static architecture with $\mathcal{O}(m)$ processes, it would take $\mathcal{O}(n^2)$ cycles per calculation. By pipelining, an actor-based architecture could carry out m calculations in the same time as a single calculation because it would initiate the next computation as soon as the replacement for the previous one had been computed— a process taking only $\mathcal{O}(n)$ steps.

Remark 2. It should also be pointed out that the structure of an actor machine is extremely concurrent: when any particular segment of the computation required by the acceptance of a communication has been completed, the resources used by the corresponding fragment of the "machine" are immediately available. It may be difficult, if one thinks in terms of

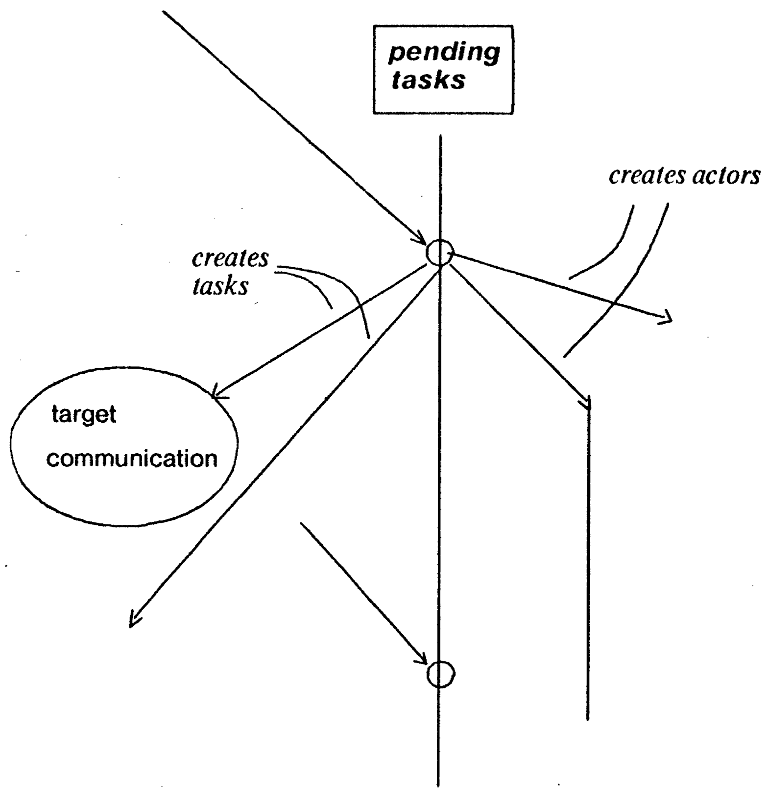

Figure 3.3: *Actor event diagrams. Each vertical line represents the events occurring in the life of an actor. The arrows represent causal links.*

sequential processes, to conceive of the inherent parallelism in the actions of an actor. The structure of computation in a sequential process is linear: typically, activations of procedures are stacked, each activation storing its current state. However, in an actor program, the absence of *assignment commands* permits the concurrent execution of the commands in a specification of the behavior of an actor. We will discuss the specific mechanisms for spawning concurrency, such as the use of *customers* to continue computations required for a transaction, later in this chapter.

3.2 Programming With Actors

In this section, we define the constructs necessary for the *kernel* of a minimal actor language. We also give some simple examples of actor programs. These examples illustrate, among other things, the versatility of message-passing as a general mechanism for implementing control structures, procedure and data abstraction in the actor construct, and the use of mail addresses instead of pointer types in data structures. The feasibility of representing control structures as patterns of message-passing was first described in [Hewitt 77].

Despite its simplicity, the kernel of an actor language is extremely powerful: it captures several important features of computation in the actor paradigm; among them, the ability to distribute a computation between concurrent elements, the ability to spawn maximal concurrency allowed by the control structure, the unification of procedural and declarative information, data abstraction and absolute containment, and referential transparency of identifiers used in a program.

An actor accepts a single communication as "input." Thus, if a computation is a function of communications from several different actors, it has to be defined using a system of actors. We will introduce linguistic constructs to simplify expressing some multi-input functions in a transparent manner. All such constructs can be defined in terms of the actors definable in a minimal actor language, and we therefore confine our present discussion to the constructs necessary for a kernel language.

3.2.1 The Basic Constructs

To define the initial configuration of an actor system we need to create some actors and to send them some communications. However, we also promote modularity by specifying the actors that may communicate with the "outside," i.e., with actors not defined within the configuration. A program in an actor language consists of:

- *behavior definitions* which simply associate a behavior schema with an identifier (without actually creating any actor).[9]
- *new expressions* which create actors.
- *send commands* which are used to create tasks.
- *receptionist declaration* which lists actors that may receive communications from the outside.
- *external declaration* which lists actors that are not part of the population defined by the program but to whom communications may be

[9]Such behavior schemas are not considered to be actors in the simple model we are currently using. In another language, such definitions can be used to create actors that are "descriptions" of actor behaviors. The behavior of such *description actors* would be to create actors of the given description when sent an appropriate communication.

sent from within the configuration.

We discuss the syntax and intended meaning for each of the expressions which can be used in a minimal language. For some simple expressions, we also show what a feasible syntax might be.

Defining Behaviors

Each time an actor accepts a communication, it computes a replacement behavior. Since each of the replacement behaviors will also have a replacement behavior, in order to specify the behavior of an actor, we need to specify a potentially infinite definition. Obviously one can not write an infinite string to define each replacement. Fortunately, we have the *principle of recursive* (or *inductive*) *definition* so familiar from mathematics. Essentially, we parameterize each expressible behavior by some identifier which will be a free variable in the definition. Whenever a behavior is specified using the behavior definition, we must specify specific values for the identifiers parameterizing the behavior definition. For example, the behavior of a bank-account depends on the balance in the account. We therefore specify the behavior of every account as a function of the balance. Whenever a particular account is created, or a replacement behavior specified, which uses the behavior definition of a bank-account, a specific value for the balance in the account must be given.

There are also an infinite number of possible values for the incoming communication. Therefore, a behavior definition is expressed as a function of the incoming communication.

Two lists of identifiers are used in a behavior definition. The first list corresponds to parameters for which values must be specified when the actor is created. This list is called the *acquaintance list.* The second list of parameters, called the *communication list*, gets its bindings from the incoming communication. When an actor is created, and it accepts a communication, it executes commands in the *environment* defined by the bindings of the identifiers.

Creating Actors

Actors are created using *new expressions* which return the mail address of a newly created actor. The mail address should be bound to an identifier or communicated; otherwise, it would not be useful to have created the actor. The syntax of new expressions would be something corresponding to the following:

$$\langle \textit{new expression} \rangle ::= \ \mathsf{new}\ \langle \textit{beh name} \rangle\ (\textit{expr}\ \{\ ,\ \textit{expr}\ \}^*\ )$$

The ⟨*beh name*⟩ corresponds to an identifier bound to a behavior given by a declaration using a *behavior definition.* A new actor is created with the behavior implied by the *behavior definition* and its parameters are instantiated to the values of the expressions in the parenthesis. In actor jargon, we have defined the *acquaintances* of an actor. The value of the expression is the mail address of the actor created and it can be bound to an identifier called an *actor name* by a ⟨*let command*⟩. An *actor name* may be used as the target of any communication, including communications sent in the initial configuration.

Actors created concurrently by an actor may know each others mail addresses. This is a form of mutually recursive definition permissible in actors. However, all the newly created actor knows is the mail address of the other actor: It does not have any other direct access to the internal structure of that actor.

Creating Tasks

A task is created by specifying a target and a communication. Communications may be sent to actors that already existed, or to actors that have been newly created by the sender. The target is the mail address of the actor to which the communication is sent. The syntax of a command that would create tasks is something like the one given below:

$$\langle \textit{send command} \rangle ::= \ \mathsf{send}\ \langle \textit{communication} \rangle\ \mathsf{to}\ \langle \textit{target} \rangle$$

where a *communication* is a sequence of expressions (perhaps empty). The expressions may be identifiers, constants, or the appropriate functions of these. The expressions are evaluated and the corresponding values are sent in the communication. The *target* is an identifier bound to the mail address of an actor.

Declaring Receptionists

Although creating actors and tasks is sufficient to specify an actor system, simply doing so does not provide a mechanism for abstracting away the internal details of a system and concentrating on the behavior as it relates to outside the actor system specified by the program. In order to simplify reasoning about the composition of independently defined and debugged systems and to permit greater modularity in a system, we allow the programmer to specify the initial set of *receptionists* for any system. The receptionists are the only actors that are free to receive communications from outside the system. Since actor systems are dynamically evolving and open in nature, the set of receptionists may also be constantly changing. Whenever a communication containing a mail address is sent to an actor outside the system, the actor residing at that mail address can receive communications from the outside and therefore *become* a receptionist. The set of receptionists increases as the system evolves.

If no receptionists are declared, the system can not initially receive communications from actors outside the system. However, the mail address of an actor may subsequently be delivered to an external actor, so that the actor system may evolve to include some receptionists. This illustrates the potentially dynamic nature of the set of receptionists.

Declaring External Actors

Communications may be sent to actors outside an actor system. Typically, an actor may get the mail address of another actor which is not in the system in a communication from the outside. It would then be able to send communications to this actor. However, even when an actor system is being defined, it may be intended that it be a part of a larger system composed of independently developed modules. Therefore, we allow the ability to declare a sequence of identifiers as *external.* The compiler associates these identifiers with actors whose behavior is to buffer the communications they accept. Whenever a given actor system is composed with another in which the external actors are actually specified, the buffered mail can be forwarded to the mail address of the actual actor (which was hitherto unknown). We will show how the compositionality can be actually implemented in an open, evolving system using message-passing.

There need be no external declaration in an program. In this case, no communication can initially be sent to mail addresses outside the actor system defined by the program. However, as the system receives communications from the outside, the set of external actors will "grow." Notice that it is useless to have an actor system which has no receptionists *and* no external actors because such an autistic system will never affect the outside world!

Commands

The purpose of commands is to specify the actions to be carried out. We have already discussed most of the basic commands which would create new actors and new tasks. We also need a command to specify a replacement. The syntax of the become command in SAL is:

become ⟨*expression*⟩

where the expression is bound to a mail address. The actor simply forwards all its mail to the actor at the specified mail address. If the expression is a *new expression*, then there is no need to assign a new mail address to the created actor since that mail address would be equivalent to the mail address of the actor it is replacing. Thus the picture in Fig. 3.2 is conceptually correct. If the expression is the mail address of an already existing actor then operationally the actor *becomes* a forwarding actor to the existing actor. In this case, the picture in Fig. 3.2, although literally correct, does not express the equivalence of the two mail queues. Denotationally, the replacement behavior is the same as the behavior of the actor to which the communication is forwarded. This denotational equivalence would not be valid in a model which did <u>not</u> assume arrival order non-determinism and the guarantee of delivery.

There is one other kind of command which is necessary: a conditional which determines which branch is taken. Conditional or branching commands are of the usual *if-then* or *case* form. It is also useful to allow *let bindings* so that identifiers may serve as a shorthand for expressions in a particular context. We have already shown the use of *let* bindings in the recording of the mail addresses of newly created actors.

Default Behaviors

Since all actors must specify a replacement behavior, we use the default that whenever there is no executable become command in the code of an actor in response to some communication, then we replace that actor with an identically behaving actor. Since the behavior of an actor is determined by a finite length script involving only conditional commands for control flow, it is can be thought of as a finite depth tree one of whose branches is executed. The particular branch executed depends on the communication.[10] Thus it is (easily) decidable if no replacement has been specified for a given acquaintance and communication list.

3.2.2 Examples

We define several examples of programs written using actors. These examples illustrate the relative ease with which various *data structures* and *control structures* can be implemented in an actor language. Specifically, we will give the implementation of a *stack* as a "linked list" of actors. This simple example also illustrates how the acquaintance structure makes the need for pointer types superfluous in an actor language. Other data structures can be defined in a similar manner.

The second example we present is that of the *recursive factorial function.* This is a classic example used in (almost) any work on actors. An iterative control structure can also be easily defined [Hewitt 77]; we leave it as an exercise for the interested reader. The technique for an iterative factorial is similar to the standard *accumulation of parameters* in functional programming. The final example in this section is an implementation for an actor specified by an external declaration. This example should clarify the use of external declarations to bind actors that are in the population of some independent module. The independent module can be later composed with the module presently being specified. We will deal with some more

[10]The tree need not be finitely branching because the communications can be one of an arbitrary countable set.

complex examples in the next chapter.

Example 3.2.1 A Stack. We implement a stack as a collection of actors with uniform behavior. These actors will represent total containment of data as well as the operations valid on such data. Assume that the linked list consists of a collection of nodes which store a value and know the mail address of the "next" actor in the link. The code for defining a stack element is given below. We skip all error handling code because such details will simply detract from the basic behavior being expressed. We assume that there is a pre-defined value NIL and use it as a bottom of the stack marker. Two kinds of operations may be requested of a *stack-node*: a *push* or a *pop*. In the first case, the new content to be pushed must be given, and in the second, the customer to which the value stored in the *stack-node* can be sent.

```
a stack-node with acquaintances content and link
   if operation requested is a pop $\wedge$ content $\neq$ NIL then
       become link
       send content to customer
   if operation requested is push then
       let P = new stack-node with current acquaintances
       { become new stack-node with acquaintances new-content and P }
```

The top of the stack is the only receptionist in the stack system and was the only actor of the stack system created externally. It is created with a NIL content which is assumed to be the bottom of the stack marker. Notice that no mail address is ever communicated by any node to any external actor. Therefore no actor outside the configuration defined above can affect any of the actors inside the stack except by sending the receptionist a communication. When a *pop* operation is done, the actor on top of the stack simply *becomes* the next actor in the link. This means that all communications received by the top of the stack are now *forwarded* to the next element.

For those concerned about implementation efficiency, notice that the underlying architecture can splice through any chain of *forwarding actors* since their mail address would no longer be known to any actor, and in due course, will not be the target of any tasks. The user is entirely free from considering the details of such optimizations.

Example 3.2.2 A Recursive Factorial. We give this classic example of a recursive control structure to illustrate the use of *customers* in implementing continuations. The example is adapted from [Hewitt 77] which provided the original insight exploited here. In a sequential language, a recursive formula is implemented using a stack of activations. In particular, the use of a stack implies that a factorial can accept only one communication from some other actor and is busy until it has computed the factorial of the given number. There is no mechanism in the sequential structure for distributing the work of computing the factorial or concurrently processing more than one request.

Our implementation of the *factorial actor* relies on creating a customer which waits for the appropriate reply, in this case from the factorial itself, so that the factorial is concurrently free to process the next communication. We assume that a communication to a factorial includes a mail address to which the value of the factorial is to be sent. The code for a recursive factorial is given below. Note that we use *self* as the mail address of the actor itself. This mail address will be instantiated when an actor is actually created using the *behavior definition* and serves as shorthand by eliminating the need for a parameter in the definition.

```
Rec-Factorial with acquaintances self
    let communication have an integer n and a customer
    become new Rec-Factorial
    if $n = 0$
        then send [1] to customer
    else let c be a Rec-Customer created which will accept an integer k
                 and send $n*k$ to the customer
             { send $n - 1$ , the mail address of c to self }
```

In response to a communication with a non-zero integer, n, the actor with the above behavior will do the following:

- Create an actor whose behavior will be to multiply the n with an integer it receives and send the reply to the mail address to which the factorial of n was to be sent.
- Send itself the "request" to evaluate the factorial of $n - 1$ and send the value to the customer it created.

One can intuitively see why the factorial actor behaves correctly, and can use induction to prove that it does so. Provided the *customer* is sent the correct value of the factorial of $n - 1$, the *customer* will correctly evaluate the factorial of n. What's more, the evaluation of one factorial doesn't have to be completed before the next request is processed; i.e., the factorial actor can be a shared resource concurrently evaluating several requests. The behavior of the factorial actor in response to a single initial request is shown in Fig. 3.4.

This particular function is not very complicated, with the consequence that the behavior of the *customer* is also quite simple. In general, the behavior of the customer can be arbitrarily complex. The actor originally receiving the request delegates most of the processing required by the request to a large number of actors, each of whom is dynamically created. Furthermore, the number of such actors created is in direct proportion to the magnitude of the computation required.

There is nothing inherently concurrent in the recursive algorithm to evaluate a factorial. Using the above algorithm, computation of a single factorial would *not* be any faster if it were done using a sequential language as opposed to an actor language. All we have achieved is a representation of the stack for recursion as a chain of customers. However, given a network of processors, an actor-based language could process a large number of requests much faster by simply distributing the actors it creates among these processors. The factorial actor itself would not be the bottleneck for such computations. (Of course, it would be useful to have fast communication

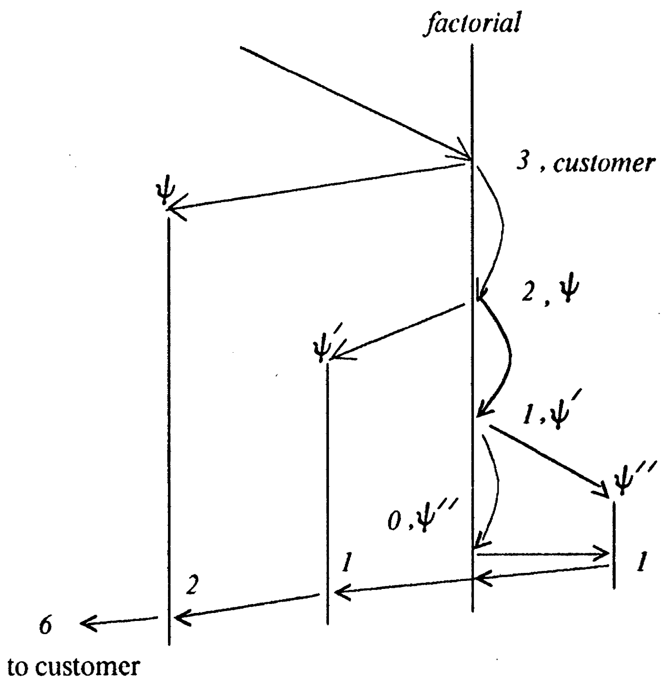

Figure 3.4: *The computation in response to a request to evaluate the factorial of 3. The ψ's represent dynamically created customers (see text).*

links between the processors).

In general, there are also more parallel algorithms for evaluating functions, and these algorithms can be exploited in an actor-based language. For example, a more parallel way of evaluating a factorial treats the problem as that of multiplying the range of numbers from 1...n. The problem is recursively subdivided into multiplying two subranges. Such an algorithm results in the possibility of computing a single factorial in *log* n parallel time.

Example 3.2.3 External Actors. An actor program defines an initial configuration with its external actors defined by an ⟨*external declaration*⟩. To promote composition of independently programmed modules, the external actors are compiled in a specific manner. This example simply illustrates how one might implement external actors. The desired behavior of an external actor is to as follows:

- simply hold all communications sent to it until the system is composed with another that contains the actor in question.
- respond to a communication telling it to forward all its mail to the actual actor when the composition is carried out.

In response to an external declaration, we actually create an actor which will exhibit the above behavior.

The code for an implementation can be given as follows. Assume that an actor called *buffer* is simultaneously created and, appropriately enough, buffers all communications until it accepts a communication telling it to forward them to a given mail address. Such a *buffer* could be specified as a queue using a linked list in a manner analogous to the implementation of the stack given above. One could also be a bit perverse and specify the buffer as a stack without changing the correctness of its behavior (recall the arrival order nondeterminism of the communications). As a stack, the behavior of the *buffer* would be given as below:

```
Buffer with acquaintances content and link
    if operation requested is release $\wedge$ content $\neq$ NIL then
        send content to customer
        send release request with customer to link
        become customer
    if operation requested is hold then
        let B be a new buffer with acquaintances content and link
        { become new buffer with acquaintances new-content and B }
```

Assume for the purposes of simplification that a protocol for specifying a communication to become the actor at the mail address *m* exists and that such a communication has the form *become m*, where *m* is the mail address of the actor to which the mail should be forwarded. The behavior of an external actor is specified as below:

```
extern with acquaintances buffer
    if the communication is become customer
        then  become customer
              send release request with customer to buffer
        else send hold request with customer to buffer
```

3.3 Minimal Actor Languages

In this section, we give the syntax for two minimal languages, SAL and *Act*. The programming language SAL has been developed for pedagogical reasons and follows an algol like syntax. *Act* is related to the languages implemented by the Message-Passing Semantics Group at M.I.T. and follows a lisp-like syntax. *Act* can be considered as a kernel for the *Act3* language [Hewitt, et al 84]. One basic difference between SAL and *Act* is in how they bind identifiers and would provide for their authentication. SAL would use conventional *type-checking* whereas *Act* uses an elaborate *description system* based on a lattice structure for reasoning with the descriptions. For the rest of the thesis we will use expressions whose syntax we have already given in the previous section. For simple examples we will use SAL's syntax. However, it is not necessary to look at the details of the syntax in this section: the only feature of SAL's syntax that the reader needs to know is that the acquaintance list is enclosed in (. . .) while the communication list is enclosed in [. . .].

Notation. The usual Backus-Naur form is used. In particular, ⟨...⟩ encloses nonterminal symbols. We use darker letters for the terminals and *id* for identifiers. {...} is used to enclose optional strings, and a superscripted $*$ indicates 0 or more repetitions of the string are permissible. When a reserved symbol, such as {, is underlined, it stands for itself and not for its usual interpretation.

3.3.1 A Simple Actor Language

We give the syntax for the kernel of SAL. Behavior definitions in a SAL program are *declarative* in the the same sense as procedure declarations in an algol-like language: behavior definitions do <u>not</u> create any actors but simply identify a identifier with a behavior template. Actors are created by *new expressions* whose syntax is the same as that given in the last section.

The syntax of *behavior definitions* is as follows:

$\langle$*behavior definition*$\rangle$ ::=
 def $\langle$*beh name*$\rangle$ ($\langle$*acquaintance list*$\rangle$) [$\langle$*communication list*$\rangle$]
  $\langle$*command*$\rangle^*$
 end def

Quite often the identifiers to be bound depend on the kind of communication or acquaintance list: For example, if the communication sent to a bank is a withdrawal request then the communication must also specify the amount to be withdrawn; but if the communication is a request to show the balance, then it should not specify any amount. We follow the *variant record* structure of Pascal [Wirth 72] to deal with the variability of the identifier bindings. Basically, we branch on the value of an identifier called the *tag-field* and depending on the value of the tag-field, different identifier bindings are expected. The value of tag-field is called a *case label.*
The syntax of the parameter lists is as follows:

$\langle$*parameter list*$\rangle$ ::= {*id* | $\langle$*var list*$\rangle$ } | { , *id* | , $\langle$*var list*$\rangle$ }* | ϵ
$\langle$*var list*$\rangle$ ::= case $\langle$*tag-field*$\rangle$ of $\langle$*variant*$\rangle^+$ end case
$\langle$*variant*$\rangle$ ::= $\langle$*case label*$\rangle$: $\langle$*parameter list*$\rangle$

where *id* is an identifier, ϵ is an empty string (in case the parameter list is empty), the *tag field* is an identifier, and the *case label* is a constant (data-value). The example below illustrates the use of parameter lists. A communication list in the behavior definition of a bank account is given.

```
case request of
    deposit     : ( customer , amount )
    withdrawal  : ( customer , amount )
    balance     : ( customer )
end case
```

Thus a communication [*deposit* , *Joe* , *$50.00*], where Joe is the mail address of some actor, would be an appropriate communication to send to a bank account created using the above behavior definition.

We avoid specifying any type structure in our programming language for the sake of simplicity. It is not difficult to specify one: All we would have to do is use type declarations with the every identifier. *Static type checking* could be performed when the code is compiled to make sure that the identifiers are used correctly in the commands (with respect to their types). For example, identifiers used as targets must have the type mail address. *Dynamic type-checking* can be used whenever a new actor is actually created: it would check if the parameters are correctly instantiated. Dynamic type-checking would also have to be used when a communication is accepted.

```
⟨command⟩ ::= if ⟨logical expression⟩ then ⟨command⟩
                    { else ⟨command⟩ } fi   |
              become ⟨expression⟩   |
              ⟨send command⟩  |  ⟨let bindings⟩ {command}
              ⟨behavior definition⟩  |  ⟨command⟩*
```

The syntax is for the most part quite obvious. We have already defined *behavior definitions* above. Note that the scope of an identifier bound by a behavior definition is lexical. The syntax for *send command* was given in the last section. It is simply:

$$\langle \textit{send command} \rangle ::= \text{send } \langle \textit{communication} \rangle \text{ to } \langle \textit{target} \rangle$$

let bindings allow one to use an abbreviation for an expression. There is no mutual recursion unless new expressions are being bound; in the latter case, the actors created can know each others mail addresses. The syntax for *let bindings* is as follows:

```
⟨let bindings⟩ ::= let id = ⟨expression⟩
                   and id = ⟨expression⟩
                   ......
```

We give only one example of a behavior definition in SAL to illustrate the flavor of the syntax. The code below is for an actor which behaves like a *stack-node* discussed in example 3.2.3 (§3.2).

```
def stack-node (content,link )
    [ case operation of
        pop : (customer)
        push : (new-content)
    end case]
if operation = pop ∧ content ≠ NIL then
    become link
    send content to customer
fi
if operation = push then
    let P = new stack-node (content,link)
    { become new stack-node (new-content , P)}
fi end def
```

Note that we assume NIL is a predefined value and SINK is the mail address of some actor. A node can be created by a *new* command of the form given below.

let p = new *stack-node* (NIL,SINK)

The node created will subsequently serve as the receptionist for the stack since the mail address bound to p will always represent the mail address of the top most node of the stack.

3.3.2 Act

The language *Act* is a sufficient kernel for the *Act3* language which is a descendant of *Act2* [Theriault 83]. One basic distinction between *Act* and SAL is that the former uses a *keyword-based* notation while the latter uses a *positional* notation. The acquaintance list in *Act* is specified by using identifiers which match a pattern. The pattern provides for freedom from *positional* correspondence when new actors are created. Patterns are used in pattern matching to bind identifiers, and authenticate and extract information from data structures. The simplest pattern is a *bind pattern* which literally binds the value of an identifier to the value of an expression in the current environment. The syntax of pattern matching is quite involved and not directly relevant to the our purposes here. We therefore skip it.

When an actor accepts a communication it is *pattern-matched* with the *communication handlers* in the actor's code and dispatched to the handler of the pattern it satisfies. The bindings for the communication list are extracted by the pattern matching as well. We do not provide the syntax for expressions except to note that the *new expressions* have the same syntax as in §3.2 namely the keyword new followed by an expression. The syntax of behavior definitions in *Act* programs is given below.

```
⟨behavior definition⟩ ::=
   (Define (new id { (with identifier ⟨pattern⟩) }*)
                ⟨communication handler⟩*)
⟨communication handler⟩ ::=
   (Is-Communication ⟨pattern⟩ do ⟨command⟩*)
```

The syntax of commands to create actors and send communications is the same in actor definitions as their syntax at the program level. The syntax of the *send-to command* is the keyword send-to followed by two expressions. The two expressions are evaluated; the first expression must evaluate to a mail address while the second may have an arbitrary value. The result of the send-to command is to send the value of the second expression to the target specified by the first expression.

```
⟨command⟩ ::= ⟨let command⟩ | ⟨conditional command⟩ |
              ⟨send command⟩ | ⟨become command⟩

⟨let command⟩ ::= (let (⟨let binding⟩*) do ⟨command⟩*)

⟨conditional command⟩ ::= (if ⟨expression⟩
                                (then do ⟨command⟩*)
                                (else do ⟨ command⟩*))
```

```
⟨send command⟩ ::= (send-to ⟨expression⟩ ⟨ expression⟩)
⟨become command⟩ ::= (become ⟨expression⟩)
```

The example of a stack-node definition from §3.2 is repeated below. For simplicity, we skip all error handling code. Note the keywords in the acquaintance and communication lists. These keywords allow a free order of attributions when the actors are created or when communications are sent. All the bindings we give are simple; in general the bindings can be restricted to complex patterns which allow authentication of the data by pattern matching.

```
(define (new stack-node (with content ≡c)
                        (with next-node ≡next))
   (Is-Communication (a pop (with customer ≡m)) do
      (if (NOT (= c empty-stack))
         (then(become next)
             (send-to (m) (a popped-top (with value ≡c))))))
   (Is-Communication (a push (with new-content ≡v)) do
      (let ( x = new stack-node (with content c)
                        (with next-node next)).
      do (become new stack-node (with content v)
                        (with next-node x)))))
```

Chapter 4

A More Expressive Language

In this chapter, we will define some higher-level constructs that make the expression of programs somewhat simpler. The purpose of this exercise is two-fold: firstly, we wish to build a somewhat richer language, and secondly, we illustrate the versatility of the constructs in a minimal actor language. For purposes of brevity, we will use SAL in simple examples. In more involved examples, we simply use pseudo-code. The issues discussed in this chapter include: developing a notation to represent functions whose arguments are supplied by communications from several different actors; the question of *delegation* which arises when determining the replacement actor requires communicating with other actors; the meaning and representation of *sequential composition* in the context of actor system; and lastly, the implementation of *delayed* and *eager* evaluation for arbitrary expressions. The interest in such evaluation strategies stems in part because they are interesting ways to demonstrate the utility of mapping values like numbers into a corresponding set of actors.

4.1 Several Incoming Communications

One of the simplest questions one can ask is what the representation of functions of several different inputs is going to be. If all the values needed to evaluate a function are to be received from the same actor, and at the same time, then there is no issue because communications in the kernel language are defined as a list of values. In general, however, carrying out some computation may require values from different actors. An actor need not know who the sender of the communication it is currently processing is. Modelling the above situation requires using some special protocols. The specifics of the construction are dependent on the type of scenario in which the multiple inputs are required.

4.1.1 A Static Topology

There are two distinct possible scenarios for an actor representing a function of several arguments. If the sender is irrelevant, then the actor simply *becomes* an actor which responds appropriately to the next incoming communication. If the senders are relevant but static, as in *dataflow* languages, then we can represent the function as a system of actors: one actor as the *receptionist* for each sender and one actor that does the final function evaluation. Each receptionist buffers communications until it receives a *ready* communication from the *function-apply* actor, and then it sends the *function-apply* actor another communication together with its own mail address. The mail addresses serve to identify the sender. A concrete picture for such a *function-apply* is an agent on an assembly line which is putting "nuts" and "bolts" together and needs one of each to arrive in order to fasten them before passing the result on. The receptionists act to buffer the "nuts" and "bolts."

Consider the simple case of a *function-apply* actor which needs two inputs and sends the result to an actor at the mail address m, as shown in Fig. 4.1. We assume actors at mail addresses m_1 and m_2 act to buffer incoming arguments and are the *receptionists* for this system of three actors. The actor at m is an external actor. The program for the actor to evaluate the function f can be given as below.

We give two mutually recursive definitions. Only one actor need be created using the *two-inputs-needed* definition. The behavior of this actor will be alternately specified by one or the other of the definitions. One observation that can be made is that the mutual recursion in the definitions is simply to make it easier to understand the code: It would be entirely possible to write a single definition to achieve the same purpose. The alternate definition would use an acquaintance and branch on its value to the two possible behaviors.

```
def $two\text{-}inputs\text{-}needed\ (m_1, m_2, m)\ [\ sender, arg\ ]$
  if $sender = m_1$
    then become new $one\text{-}input\text{-}needed\ (m_1, m_2, second, arg)$
    else become new $one\text{-}input\text{-}needed\ (m_1, m_2, first, arg)$
  fi end def
```

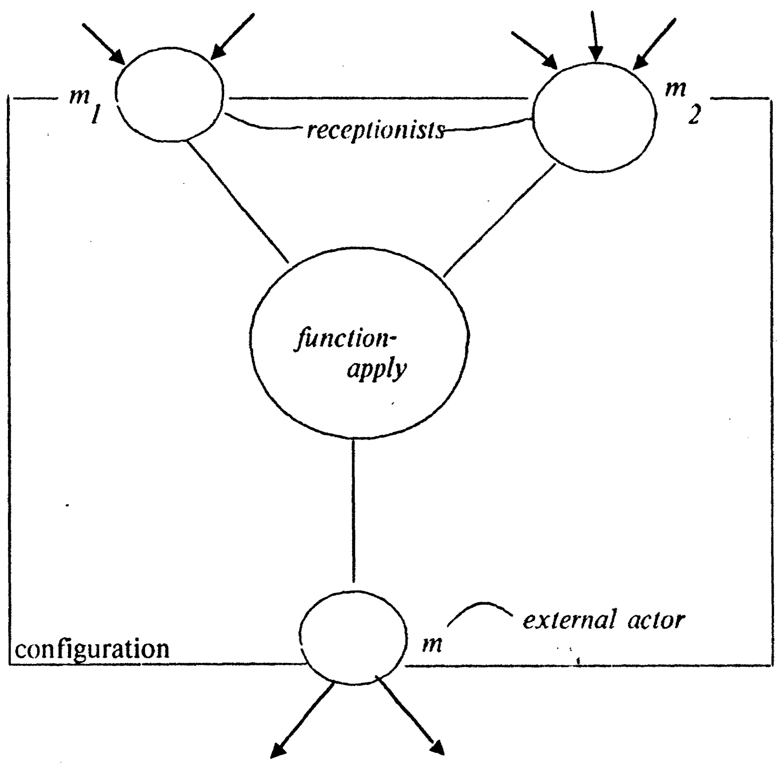

Figure 4.1: *A fixed topology for a two input function.*

```
def one-input-needed ($m_1$, $m_2$, $m$, new-arg-position, old-arg)
                    [ sender , arg ]
    let $k$ = ( if new-arg-position = second then $f$ (old-arg , new-arg)
              else $f$ (new-arg , old-arg) fi )
                    { send [ $k$ ] to $m$ }
    send ready to $m_1$
    send ready to $m_2$
    become new two-inputs-needed ($m_1$ , $m_2$)
end def
```

The *function-apply* actor which needs two inputs from actors m_1 and m_2 can be created by the expression new *two-inputs-needed* (m_1, m_2). We assume that the actor m is defined in the lexical scope of the new expression.

4.1.2 A Dynamic Topology

A more interesting case of a many argument function is one in which the senders can vary. One frequently useful form occurs when more input to complete some computation may depend on the segment of the computation that has been carried out so far. Such a situation represents a dynamic topology of the interconnection network of actors. For example, an interactive program may need more input to continue with some transaction. The source of the input may vary: the program may sometimes get the input off some place on a disk, or perhaps from a magnetic tape, or a user. A static topology where all the communications are received from the same senders before the computation starts, or even during it, will not work in this case.

The general form for implementing requests for input from some particular actor is a *call expression*, which has the syntax:

$$\text{call } g\,[\,k\,]$$

where k is a communication and g is an identifier bound to a mail address. The value of the call expression is the communication sent by g as the *reply* when it accepts the present communication k. One way to picture the flow of the computation is given in Fig. 4.2. However, the figure is somewhat misleading as a representation of what actually occurs in an actor system. The actor f does not (necessarily) have to wait for the reply from the actor g: a customer can be created which will continue processing when the *reply* from the actor g arrives. While the customer is "waiting" for the reply from g, the actor f may accept any communications pending in its queue.

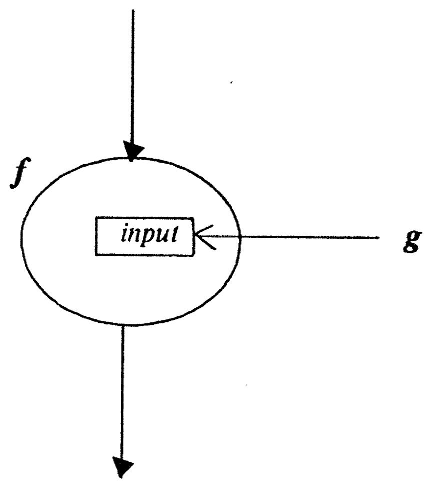

Figure 4.2: *The behavior of actor f in response to a communication may be a function of a communication from the actor g.*

The use of customers to implement *continuations* is more accurately portrayed in Fig. 4.3. This figure may be compared to the example of the recursive factorial in §3.2. There is some sequentiality, modeled by the causality ordering of the events, in the course of the computation triggered by a communication to the actor f. There is a degree of concurrency as well. If the call expression occurs in the following context in the code for f:

$$S' \quad \text{let } x = (\text{ call } g[k]\,)\ \{S\} \quad S''$$

then the actions implied by S' and S'' can be executed concurrently with the request to g. Moreover, as discussed above, we do not force the actor f to wait until the reply from the actor g is received. The actor f would be free to accept the next communication on its mail queue, provided it can compute its replacement.[1] The customer created to carry out the actions implied by the command S will wait for the reply from the actor g.

Notice that the general scheme for representing requests is analogous to our earlier implementation of the factorial actor. Using a *call* expression, the program for a recursive factorial may be written as below:

```
def exp Rec-Factorial ( ) [ $n$ ]
    become new Rec-Factorial ( )
    if $n = 0$
        then reply [1]
        else reply [ $n$ * (call self [$n - 1$] ) ]
    fi end def
```

We use def exp instead of def so that it is clear that the actor will return a reply to a customer that is implicit in all communications accepted. The incoming communication will have the form:

[1] We will discuss the case where an actor can not compute its replacement without further input in the next section.

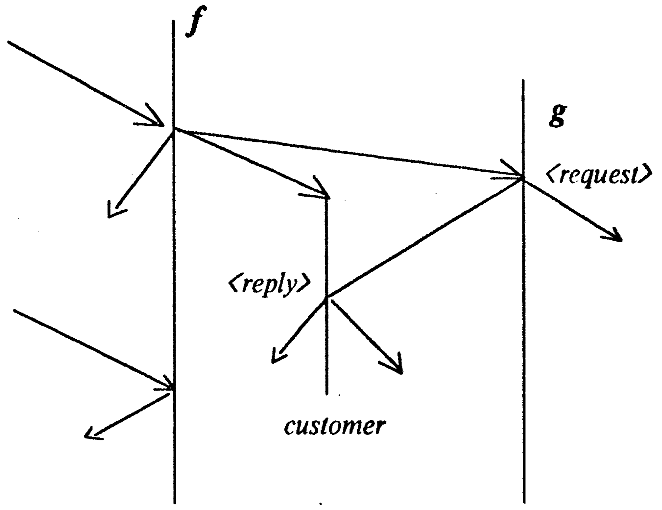

Figure 4.3: *The behavior of actor f is defined by program with a* **call** *expression which requests more input. Some of the events are activated by the reply to a customer.*

$$[\ m\ ,\ k_1, \ldots, k_j\ ]$$

but our syntax explicitly shows only $[k_1, \ldots, k_j]$. The mail address m is bound when the expressional actor gets a communication. A *translator* can insert the customer and subsequently map the command reply $[\ x\ ]$ into the equivalent command:

$$\textsf{send}\ [\ x\ ]\ \textsf{to}\ m$$

The actor at m will be the customer which will continue the transaction initiated at the time of its creation. Comparing the above code with that of factorial in the previous chapter (see Fig. 3.4) should make it clear how the behavior of the appropriate customer can be deduced: essentially, the segment of the environment which is relevant to the behavior of the customer has to preserved; a dynamically created customer can do this. A SAL compiler whose target language is the kernel of SAL can translate the above code to one in which the customer creation is explicit. Also note that only one reply command may be executed (in response to a single request).

Thus a purely expression oriented language can be embedded in SAL (or equivalently in *Act*). The concurrency in such a language is inherent and the programmer does not have to worry about the details related to creating customers for implementing continuations. Another advantage to the "automatic" creation of customers is that it provides protection against improper use by the programmer, since the programmer has no direct access to the mail address of the customer created.

There is one aspect of the expression oriented language that may be disturbing to the functional programming *aficionados*: namely, the presence of side-effects implicit in the become command. Recall that the ability to specify a replacement behavior is necessary to model objects with changing local states. The become command provides a mechanism to do so. The become command is actually somewhat analogous to recursive feedback in a *dataflow* language. This similarity (and the differences) will be discussed in greater detail in chapter 6.

4.2 Insensitive Actors

When an actor accepts a communication and proceeds to carry out its computations, other communications it may have received must be buffered until the replacement behavior is computed. When such a replacement actor is known, it processes the buffered communications, as well as any new ones received. The precise length of time it takes for an actor to respond to a communication is not significant because no assumption is made about the arrival order of communications in the first place.[2]

However, the desired replacement for an actor may depend on communication with other actors. For example, suppose a checking account has overdraft protection from a corresponding savings account. When a withdrawal request results in an overdraft, the balance in the checking account after processing the withdrawal would depend on the balance in the savings account. Thus the checking account actor would have to communicate with the savings account actor, and more significantly the savings account must communicate with the checking account, before the new balance (and hence the replacement behavior) is determined. The relevant communication from the savings account can <u>not</u> therefore be buffered until a replacement is specified!

We deal with this problem simply by defining the concept of an *insensitive* actor which processes a type of communication called a *become communication*. A become communication tells an actor its replacement

[2]Communication delays are an important performance issue for a particular realization of the abstract actor architecture. Our focus here is restricted to semantic questions.

behavior. The behavior of an insensitive actor is to buffer all communications until it receives a communication telling it what to become. Recall that *external declarations* were similarly implemented in Example 3.2.3.

First consider what we would like the behavior of a *checking account* to be: if the request it is processing results in an overdraft, the checking account should request a withdrawal from its *savings account.* When a reply to the request is received by the checking account, the account will do the following:

- Reply to the customer of the (original) request which resulted in the overdraft; and,

- Process requests it subsequently received with either a zero balance or an unchanged balance.

Using a *call expression*, we can express the fragment of the code relevant to processing overdrafts as follows:

```
let r = (call my-savings [ withdrawal , balance − amount ])
   { if r = withdrawn
         then become new checking-acc(0, my-savings)
         else become new checking-acc (balance , my-savings)
     fi
     reply [r] }
```

To show how a call expression of the above sort can be expressed in terms of our kernel, we give the code for a bank account actor with overdraft protection. Again the code for the customers and the insensitive actors need not be explicitly written by the programmer but can instead be generated by a *translator* whenever a call expression of the above sort is used. That is to say, if a become command is in the lexical scope of a *let expression* that gets bindings using a *call expression*, then the *translator* should do the work explicitly given in the example below. Not requiring the programmer to specify the behavior of the various actors created, such as the insensitive bank account and the customer to process the overdraft, protects against erroneous communications being sent to these actors. It also frees the programmer from having to decide her own protocols.

A bank account with an overdraft protection is implemented using a system of four actors. Two of these are the actors corresponding to the checking and savings accounts. Two other actors are created to handle requests to the checking account that result in an overdraft. One of the actors created is simply a buffer for the requests that come in the checking account while the checking account is *insensitive.* The other actor created, an *overdraft process*, is a customer which computes the replacement behavior of the checking account and sends the reply to the customer of the withdrawal request. We assume that the code for the savings account is almost identical to the code for the checking account and therefore do not specify it here. The structure of the computation is illustrated by Fig. 4.4 which gives the actor event diagram corresponding to a withdrawal request causing an overdraft.

The behavior of the checking account, when it is not processing an over-

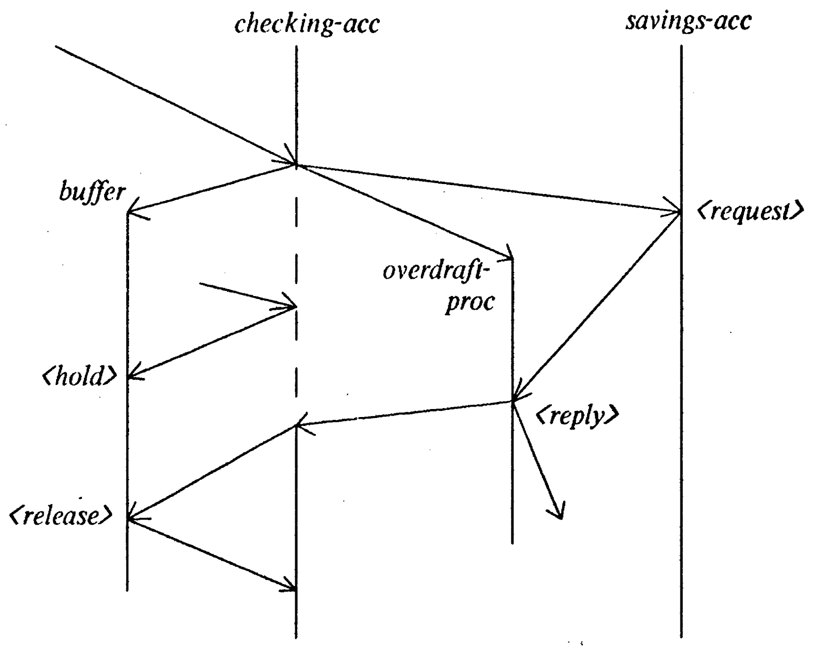

Figure 4.4: *Insensitive actors. During the dashed segment the insensitive checking account buffers any communications it receives.*

draft, is given below. When the checking account accepts a communication which results in an overdraft, it becomes an insensitive account.

```
checking-acc (balance , my-savings) [⟨request⟩]
   if ⟨deposit request⟩
     then become new ⟨checking-acc with updated balance⟩
     send ⟨receipt⟩ to customer
  if ⟨show-balance request⟩
     send [balance] to customer
  if ⟨withdrawal request⟩ then
     if balance ≥ withdrawal-amount
          then become new ⟨checking-acc with updated balance⟩
               send ⟨receipt⟩ to customer
          else let b = new buffer
               and p = new overdraft-proc
               {become new insens-acc (b,p) }
                send ⟨withdrawal request with customer p⟩ to my-savings}
```

The behavior of an "insensitive" bank account, called *insens-acc*, is quite simple to specify. It is given below. The insensitive account forwards all incoming communications to a buffer unless the communications is from the overdraft process it has created.[3] The behavior of a buffer is similar to that described in Example 3.2.3. The buffer can create a list of communications, until it receives a communication to forward them. It then forwards the buffered communications and becomes a forwarding actor so that any communications in transit will also get forwarded appropriately.

[3]Due to considerations such as *deadlock*, one would program an insensitive actor to be somewhat more "active" (see §6.1). Good programming practice in a distributed environment require that an actor be *continuously available*. In particular, it should be possible to query an insensitive actor about its current status.

```
insens-acc (buffer, proxy) [request , sender]
        if request = become and sender = proxy
              then become ⟨replacement specified⟩
              else send ⟨communication⟩ to buffer
```

Finally, we specify the code for a customer to process overdrafts. This customer, called *overdraft-process* receives the reply to the withdrawal request sent to the savings account as a result of the overdraft. The identifier *self* is bound, as always, to the mail address of the actor itself (i.e., the actor whose behavior has been defined using the given behavior definition). The response from the savings account may be a *withdrawn*, *deposited*, or *complaint* message. The identifier *proxy* in the code of the insensitive account represents the mail address of the over-draft process. The proxy is used to authenticate the sender of any *become* message targeted to the insensitive actor.

```
overdraft-proc (customer , my-checking , my-savings ,
                       checking-balance) [⟨savings-response⟩]
   send [become , self] to my-checking
   send [⟨savings-response⟩] to customer
   if ⟨savings response is withdrawn⟩
        then become new checking-acc (0 , my-savings)
        else become new checking-acc(checking-balance , my-savings)
```

4.3 Sequential Composition

In the syntax of our kernel language, we did not provide any notation for sequential composition of commands. The omission was quite intentional. Although sequential composition is primitive to sequential machines, in the context of actors it is generally unnecessary. Recall that the primitive actor carries out only three sorts of actions: namely, sending communications, creating actors, and specifying a replacement behavior. The order of these actions is immaterial because there is no changing local state affecting these actions. Furthermore, the order in which two communications are sent is irrelevant because, even if such an order was specified, it would not necessarily correspond to the order in which the communications were subsequently received.[4]

There are some contexts in which the order of evaluation of expressions seems sequential even in the kernel of SAL. The two obvious places are *conditional expressions* and *let expressions.* A *conditional expression* must be evaluated before any of the commands in the body can be executed. Such evaluation can not be done at compile time. However, the entire conditional command can be executed concurrently with any other commands at the same level. One can think of each command as an actor to which a communication is sent with the current bindings of the identifiers. The "command actor" in turn executes itself in the environment provided by the communication.

[4]Unless the two communications are sent to the same target, there may not be a unique ordering to their arrival. See the discussion in Section 2.2.

A *let command*, unless it is binding a new expression, is nothing more than an abbreviation that can be removed by the compiler if desired. A *translator* can substitute the expression for the identifier whereever the identifier is used (in the scope of the let binding).

A more interesting case is that of *let commands* binding *new expressions.* New expression bindings serve as abbreviations for behaviors instead of values. However, the behavior associated with an identifier is not necessarily constant. In an abstract sense, the identifier (in its scope of use) always denotes the same object. For example, a bank account refers to the same bank account even though the behavior of the bank account is a function of the balance in it.

Let bindings have another characteristic: They may be mutually recursive since concurrently created actors may know of each other. The question arises in what sense the behavior of an actor depends upon the other actors. The only requirement is that concurrently created actors may know each others mail address. This in turn means that the mail addresses of each of the actors should be known before any of the actors are actually created (since the behavior of each is dependent on *other* actors' mail addresses). The operational significance of this is quite straight-forward.

Not withstanding their absence in the kernel of our actor language, sequential composition of commands can be meaningful as a structural representation of certain patterns of computations. Sequential composition in these cases is a result of causal relations between events. For example, consider the commands S_1 and S_2 below:

$$S_1 \equiv \text{send } [\text{ call } g\ [x]\ ] \text{ to } f$$
$$S_2 \equiv \text{send } [\text{ call } g\ [y]\ ] \text{ to } f$$

then the sequential composition of S_1 with S_2 has a very different meaning than the concurrent composition of the two commands because the effect of accepting communication $[x]$ may be to change the actor g's subsequent behavior. Thus sequential composition can result in only some of the possible order of events inherent in the concurrent composition.

Sequential composition of the above kind is also implemented using customers. The command $S \equiv S_1 ; S_2$ is executed concurrently with other commands at the same level. To execute S, the actions implied by the command S_1 are executed, including the creation of a customer to handle the reply from g. When this customer receives the reply from g, it carries out the other actions implied by S_1 as well executing S_2.

Notice however that if S_1 and S_2 were commands to simply send communications to g, then no mechanism for any sequential composition of the two actions implied would be definable in our kernel language. Nothing signals the end of any action at an actor other than the causal relations in the events. For example, causality requires that the actions of an actor must follow the event that creates it. The conclusion to be drawn is that *concurrent composition* is <u>intrinsic</u> in a fundamental and elemental fashion to actor systems. Any *sequentiality* is built out of the underlying concurrency and is an <u>emergent</u> property of the causal dependencies of events in the course of the evolution of an actor system.

4.4 Delayed and Eager Evaluation

In this section, we will develop the model of actors in which all expressions, commands and communications are themselves considered to be actors. We will call this model of actors the *universe of actors* model. The universe of actors model is useful for defining a language that is the actor equivalent of a purely expressional language. Specifically, the universe of actors model permits an easy (and efficient) implementation of the various expression evaluation mechanisms, such as *delayed* and *eager* evaluation, using message-passing.

Computation in actor systems is initiated by sending communications to actors that are receptionists. A single behavior definition in fact represents a specification of a system of actors with one of them as the receptionist for the system; the behavior of this receptionist is to execute a sequence of commands concurrently. We can consider each command to be an actor and the receptionist, upon accepting a communication, sends each command a message to execute itself with the current environment specified by the communication sent. The command will in turn send communications to expressions and create customers to process the replies. This process must, naturally, be bottomed out at some point by actors which do not send any "requests" to other actors but simply produce "replies." Hence, we need a special kind of actor, called a *primitive actor*, with the characteristic that some of these primitive actors need not (always) rely on more *message-passing* to process an incoming communication. Furthermore, primitive actors have a pre-defined behavior which never changes (i.e., the behavior is *unserialized*). Which actors are defined as primitive depends on the particular actor system.

4.4.1 Primitive Actors

Primitive actors are used in order to "bottom-out" a computation.[5] Hence, the set of primitive actors must include the primitive data values and the basic operations on them. In particular, simple data objects such as integers, booleans and strings must be considered primitive. When an integer is sent a message to "evaluate" itself, it simply replies with itself. To carry out any computation, primitive operations, such as addition, must be pre-defined. There are various mechanisms by which a consistent model, incorporating primitive operations, can be developed: one such scheme is to also define operations such as addition to be primitive actors.

Our bias, however, is to encapsulate data values and the operations valid on the data into uniform objects. Hence, we define each integer as an actor which may be sent a *request* to add itself to another integer. The integer would then reply with the sum of the two integers. In fact an integer, n may be sent a request to add itself an arbitrary integer expression, e. In this case one must also send the local environment (which provides the bindings for the identifiers in e). The bindings of the identifiers will, of course, be primitive actors. One way to understand this notion is to notice that the expression e is really equivalent to call e [env] where env is the environment

[5] Theriault [83] used the term ***rock-bottom actors*** to describe these actors and the material on primitive actors closely follows his implementation in *Act2*.

in which the evaluation of the expression is to be performed. If e is an integer constant, it will reply with itself and, subsequently, n will reply with the correct sum. Specifically, the behavior of the expression $e + n$, in response to a request to add itself to the expression e in the environment env, can be described as:

```
let $x$ = call $e$ [ $env$ ]
        { reply [ $n + x$ ] }
```

If e is not an integer but an integer expression, a call to it must result in an integer. Thus the meta-circular behavior of the expression, $e \equiv e_1 + e_2$, is to send *evaluate* messages to each of the expressions e_1 and e_2 and to then send a message to the first expression (which would now have evaluated to the primitive actor that corresponds to the value of e_1) to add itself to the actor the second expression evaluates to.

Notice that we use integers, and expressions, as though they were identifiers bound to mail addresses, and, indeed, as actors they are. To understand this concept, consider the relation between the numeral 3 and the number 3. For our purposes, in the universe of actors model, the *identifier* 3 is bound to the mail address of the *actor 3*. Since *3* is a primitive actor, its behavior is pre-defined. Furthermore, the behavior of the actor *3* never changes (such a behavior is called an *unserialized*).

There may be more than one actor *3* in a program: the identifier 3 is completely local to the scope of its use. However, the identifier 3 has been reserved for a particular functional (unserialized) behavior and may not be used differently by the programmer. One useful implication of the fixed behavior of an integer like 3 is that it does *not* really matter how many 3's there are in a given actor system, or whether two 3's in an actor system refer to the same actor *3* or different ones. Ergo, when a communication contains the actor *3*, it is an implementation decision whether to "copy" the mail address of the actor *3* or whether to copy the actor itself: the latter possibility is useful for maintaining locality of reference in message-passing for efficiency reasons.[6] To put it another way, the unserialized nature of primitive actors implies that there is no theoretical reason to differentiate between the expression *new* 3, and simply 3.

4.4.2 Delayed Evaluation

In functional programming, *delayed evaluation* is useful for processing infinite structures by exploring at any given time, some finite segments of the structure. Using delayed expressions, the evaluation of a function is explicitly delayed until another function "resumes" it. Thus, delayed evaluation is the functional equivalent of co-routines [Henderson 80].

In actor systems, it is *not* necessary to define delayed evaluation as a primitive: Since an actor *becomes* another actor as a result of processing a task, an actor already represents an infinite structure which unfolds one step at a time (in response to each communication accepted). Similarly, co-routines are one particular case of a concurrent control structure; actors allow one to define *arbitrary* concurrent control structures. Each control

[6]There is no notion of *copying* actors in the actor model. What we mean is create a new actor with the behavior identical to the current behavior of the (old) actor.

structure defines a graph of activations of processes and, as such, every control structure can be represented as a pattern of message-passing [Hewitt 77]. The actor model allows dynamically evolving patterns of message-passing. Static control structures, such as co-routines, are a special (degenerate) case of the dynamic structures.

As the above discussion suggests, delayed evaluation is a *syntactic* extension to an actor language and not a *semantic* one. We define delayed expressions in order to make our purely expression oriented extension of SAL more expressive. The construct does not add any expressive power to the language.

The expression delay e denotes the mail address of the expression e as opposed to the actual value of e. Recall that the expression e is equivalent to call $e[env]$ where an expression denotes the mail address at which the expression resides (see the discussion about the universe of actors model in the previous section).

For purposes of the discussion below, we assume that the environment is sent to any expression receiving a request. Now we have to decide what is meant by expressions which contain delayed expressions as subexpressions. For example, the expression:

$$e_1 \equiv e_2 * \text{delay } e_3$$

is a product of an arithmetic expression and a delayed (arithmetic) expression. When e_2 has been evaluated it receives the request $[*, \text{delay } e_3]$, where delay e_3 represents the mail address of the expression e_3. Assume e_2 has evaluated to some integer n. The only feasible way of handling the expression e_1 then is to "return" (i.e., to reply with) its current local state, which will be equivalent to the expression $n * e_3$. That is exactly what is done, except that the mail address of the expression e_1 is returned. e_1 has now *become* an actor behaviorally equivalent to the expression $n * e_3$, and not the value of the expression.

4.4.3 Representing Infinite Structures

The delayed expressions we have defined so far do not really represent potentially infinite structures, because the expressions they define are not recursive. However, our def exp behavior definitions already provide for such recursive structures. In this section we explore this analogy with the help of a detailed example. We will present an example using a functional programming notation and using actors. Two different actor systems are defined with equivalent observable behavior; the second system uses actors that change their behavior. Furthermore, the second actor system does not use the list construction and separation operators. Thus the flavor of the two actor systems is quite different even though they have similar behaviors.

The Example in Functional Programming

The purpose of the following example is to define some functions which evaluate a given number of initial elements of an infinite list. The notation uses a functional form for the *cons* operation but not for the *car* or *cdr*. All functions are taken from Henderson [80]. Consider, the delayed expression in the function *integersfrom*(n) below:

$$\mathit{integersfrom}(n) \equiv \mathit{cons}(n\ ,\ \text{delay}\ \mathit{integersfrom}(n+1))$$

integersfrom(n) is an example of such an infinite list, namely the list of all the integers greater than n. This list of may be evaluated only partially at any given point in time. The function $first(i,x)$ defined below gives the first k arguments for an infinite list x whose *cdr* has been delayed. (In the functional program, one has to explicitly *force* the evaluation of a delayed list.)

```
$\mathit{first}(i,x) \equiv$ if $i=0$ then NIL
                    else cons (car $x$ , $\mathit{first}$ ($i-1$ , force cdr $x$))
```

Now we define two more functions which can be used to return the cumulative sum of all the elements of a list up to some $\underline{i^{th}}$ element. The function $sums(a,x)$ returns a list whose $\underline{i^{th}}$ element is the sum of the first i elements of the list x and the integer a. Finally, the function *firstsums*(k) uses the functions defined so far to return the list of initial sums of the first i positive integers.

```
$\mathit{sums}\ (a,x) \equiv$ cons ($a +$ car$x$ , delay( $\mathit{sums}$ ($a +$ car$x$ , force cdr $x$) )
$\mathit{firstsums}\ (k) \equiv \mathit{first}\ (k\ ,\ \mathit{sums}(0,\mathit{integersfrom}(1)))$
```

A System of Unserialized Actors

Let us now define an actor system which produces the same behavior. We will do this in two different ways. First, we define a system of actors all of whom have unserialized behaviors (i.e., they are always replaced by an identically behaving actor). We therefore give their definitions without any become commands in them. (Recall the default that an actor is replaced by an identically behaving actor if no become is found in its code). We will subsequently define a system of actors which uses serialized behaviors when appropriate. The idea behind defining two systems is to show the relation between actor creation and actor replacement. The systems also show the relation between *delay* and actor creation.

Assume that the operations cons, car and cdr exist and are defined on actors representing *lists.* cons is sent the mail address of two actors and returns a list of the two mail addresses. It is important to note the equivalence of the mail address of a primitive actor and the actor itself. There are two possibilities for a list x: it may consist of a primitive actor (equivalently the mail address of a primitive actor) or it can be the mail address of an arbitrary list. car x equals x if x is a primitive actor, or equivalently the mail address of a primitive actor, otherwise car x is the mail address of the first element of the list. cdr x is NIL if x is a primitive actor, and otherwise returns a mail address corresponding to the rest of the list.

All the actors whose behavior is given by code below are expressions. We will not bother to enclose the definitions in def exp $\cdots$ end def since the definitions are all rather brief. There is no need *delay* or *force* operators: a delayed list is represented by the mail address of an actor representing that list.

The first function we define is *integersfrom*(n). The behavior of an *integersfrom*(n) actor is that it responds to an *evaluate* request (i.e., a request of the form [ ]) by replying with a list whose car is the integer n and whose cdr is the mail address of an actor with the behavior *integersfrom*$(n+1)$.

```
integersfrom(n) [ ] ≡ reply [cons (n, new integersfrom(n + 1))]
```

The behavior of an actor whose behavior is given by *first* ( ) is as follows: when it is sent a request $[i, x]$, where i is an non-negative integer and x is an arbitrary list, it replies with the first i elements of the list. We assume that the list x is sufficiently long to have i elements.

```
first( ) [i,x] ≡ if i=0 then reply [ NIL ]
                  else reply [cons (car x , call self [i − 1, cdr x])]
```

Finally, we give the behavior definitions for the two remaining actors. *firstsums*( ) defines an actor whose behavior is to give a finite list whose $i^{\underline{th}}$ element is the sum of the first i non-negative integers. The length of the list of sums in the reply is specified in the communication received. In order to create a system which returns the list of initial sums of non-negative integers, we need to create only a *firstsums*( ) actor; all the other actors will be created by this actor. The actor created will always be the sole receptionist for such a system since no mail address is ever communicated to the outside.

```
sums(a, x ) [ ] ≡ let b = a + carx
                  { reply [ cons (b,new sums(b,cdr x))] }
```

```
firstsums( ) [k] ≡ let p = new integersfrom(1)
                   and s = new sums(0, p)
                   and f = new first( )
                   { reply[call f [k, s]] }
```

The fact that all the behaviors are unserialized implies that it is possible to use the same actors for different requests. Thus if an actor with behavior *first*( ) exists, it doesn't matter if a communication is sent to the same actor or to a new actor created with the behavior *first*( ). The converse of this property is that an actor with unserialized behavior can never be a *history-sensitive shared object.* This same limitation is applicable to purely functional programs.

A System With Serialized Actors

We now attack the same problem with actors that may change their local state: i.e., actors that may be replaced by actors whose behavior is different than their own. The point of defining this system is to show the relation between actor creation and replacement. The example also illustrates the similarity between a *delayed expression* and a *serialized actor.*

It should be noted that actors are in fact more general than expressions in functional programming. For one, actors, unlike expressions, may represent (history-sensitive) shared objects. For example, a bank account written as a function which returns a partly delayed expression will have returned an argument purely local to the caller. This means that such a bank account can not be shared between different users (or even between the bank manager and the account owner!). In dataflow architectures, the problem of sharing is addressed by assuming a special *merge* element. However dataflow elements have a static topology (see the discussion in chapter 2).

The definitions below do not use *cons*, *car*, and *cdr* operations. Instead we simply construct and bind the communication lists. The behavior definition of *integersfrom(n)* is that it accepts a simple evaluate message, [ ], and replies with the integer n. However, the actor presently *becomes* an actor with the behavior *integersfrom(n+1)*. An actor with its behavior defined by *sums(a,x)* has two acquaintances, namely a and x. a is the sum of the first *umpteen* elements and x is the mail address of an actor which replies with the *umpteen+1* element of the "list." The *sums* actor calls x and replies with the next sum each time it is called.

The behavior definitions of *first* is similar to the previous section except that we use explicit *call*'s. Note that the definition of *firstsums*( ) is identical to the one given above, and is therefore not repeated.

```
integers-from(n) [ ] ≡ reply [ n ]
                       become new integers-from(n + 1)

first( ) [i, x] ≡ if i=0 then reply [ ]
                  else reply [call x [ ], call self [i − 1, x] ]

sums(a, x) [ ] ≡ let b = a + call x [ ]
                 {reply [ b ]
                 become new sums(b, x) }
```

The concept of *replacement* provides us with the ability to define *lazy evaluation* so that same expression would not be evaluated twice if it was passed (communicated) unevaluated (i.e., if merely its mail address was sent). If lazy evaluation was desired, one could send communications containing the mail addresses of expressions, instead of the primitive actors the expressions would evaluate to. In this scheme, the message-passing discipline is equivalent to a *call-by-need* parameter passing mechanism, instead of a *call-by-value* which is the default in our definition of SAL.

However, the point of actor architectures is not so much to merely conserve computational resources but rather to provide for their greedy exploitation— in other words, to spread the computation across a extremely large-scale distributed network so that the overall parallel computation time is reduced. At the same time, it would be inadvisable to repeat the same computation simply because of the lack of the ability to store it— a serious problem in purely functional systems [Backus 77]. In the next section we provide a strategy for evaluation of expressions which satisfies these requirements.

4.4.4 Eager Evaluation

The inherent parallelism in actors provides many options for a greedy strategy in carrying out computations. The idea is to dynamically spawn numerous actors which will carry out their computations concurrently. These actors can exploit all the available resources in a distributed systems. We have already seen pipelining of the replacement actors as a mechanism for increasing the speed of execution on a parallel architecture. In an actor language, the pipelining is made possible by the use of customers by which *continuations* are incorporated as first-class objects.

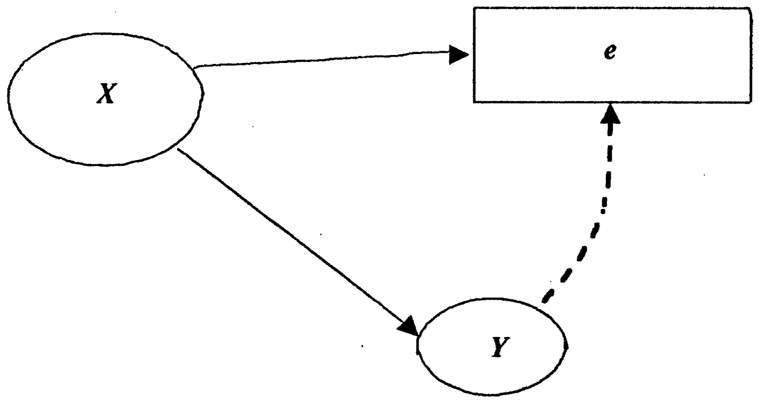

Figure 4.5: *Eager evaluation. The dotted line shows the acquaintance relation. X creates Y and tells it about e while concurrently sending an evaluate message to e*

Another mechanism by which the available parallelism in an actor language can be exploited is by schemes for *eager evaluation.* To speed up the computation to its logical limits, or at least to the limit of the number of available processes in a particular network, one can create an actor with the mail addresses of some expressions (which have not necessarily been evaluated) as its acquaintances. So far, this is similar to how one would implement *call-by-need.* However, for eager evaluation we concurrently send the expression, whose mail address is known to the actor created, a request to evaluate itself. Fig. 4.5 shows this pictorially. The net effect is that an actor Y which has been created may accept a communication even as the expression e which is its acquaintance is being evaluated concurrently. The expression subsequently *becomes* the primitive actor it evaluates to. Thus the evaluation of the same expression need not be repeated.

Chapter 5

A Model For Actor Systems

A model for any collection of objects provides a map from the objects into equivalence classes that contain elements which are considered to be indistinguishable from each other. In other words, a *model* provides an abstract perspective in which the "irrelevant" details are ignored in establishing the equivalence of systems. A *denotational model* is one in which the meaning of a system can be derived from the meanings of its constituent parts. We will refer to this property as *compositionality.*

The semantics of sequential programming languages has been rather successful in building *denotational models* of programs which abstract away the *operational* details of the sequential systems defined by the programs. In the case of concurrent systems, however, the requirements of compositionality have resulted in proposed denotational models which retain substantial operational information. The reason for this is as follows. Composition in concurrent systems is achieved by inter-leaving the actions of the systems that are composed: thus the denotations for a system require the retention of information about the intermediate actions of the system (see, for example, [Milner 80] or [de Bakker and Zucker 83]).

In this chapter we will develop a model for actor systems based on semantics by reductions. The actor semantics follows a structured operational style long advocated by Plotkin. In particular, we define transition relations which represent the evolution of an actor system as the computations it is carrying out are unfolded. Two transition relations are necessary to capture the behavior of an actor system. The first of these, called a *possible transition*, represents the possible orders in which the tasks may be processed. The possible transition relation is, however, insufficient to capture the guarantee of mail delivery. We therefore define a second transition relation, called *subsequent transition*, which expresses just such a guarantee.

The plan of this chapter is as follows. The first section specifies a formal definition for the configuration of an actor system and states the requirements relevant to defining an operational semantics of actors. In the second section we map actor programs to the initial configurations they define. The last section discusses two kinds of transition relations between configurations. These transition relations provide an operational meaning to actor programs.

5.1 Describing Actor Systems

The *configuration* of an actor system is described by the actors and tasks it contains. There is no implied uniqueness in the configuration of an actor system: different observers may consider the system to be in quite different configurations. This issue is discussed in greater detail in Section 5.3. To describe the actors in a system, we have to define their behaviors and their topology. Descriptions of actor systems are embodied in configurations and therefore we will first develop some notation to represent configurations. The definitions below assume that actor behaviors are well-defined— a topic we will discuss in §5.2.

5.1.1 Configurations

There are two components in a configuration: namely, the actors and the tasks. The tasks represent communications which are still pending; in other words, communications that have been sent but not yet accepted by the target. These communications may or may not have been delivered; they are simply yet to be processed. We keep equivalent tasks (i.e., those with the same communication and target) distinct by specifying a unique tag for each task in a configuration.

Definition 5.1 Tasks. *The set of all possible tasks, $\mathcal{T}$, is given by*

$$\mathcal{T} = \mathcal{I} \times \mathcal{M} \times \mathcal{K}$$

where $\mathcal{I}$ is the set of all possible tags, $\mathcal{M}$ is the set of all possible mail addresses, and $\mathcal{K}$ is the set of all possible communications. We represent *tags and mail addresses as finite sequences of natural numbers, separated by periods, and communications as a tuple of values. If τ is a task and $\tau = (t, m, k)$ then we call t the tag for the task τ and m the target.*

We define a *local states function* to represent the behaviors of the actors from some view-point. Since there are only finitely many actors in any given configuration, this is really a partial function on the set of all possible mail addresses. However, when appropriate, one can treat the *local states function* as a total function by defining an *undefined behavior*, called $\perp$, and mapping all undefined elements to $\perp$. For our immediate purposes, defining a total function is not necessary. In the definition below, we assume that a set of possible actor behaviors $\mathcal{B}$ exists.

Definition 5.2 Local States Function. *A local states function l is a mapping from the mail addresses of the actors in a system to their respective behaviors, i.e.,*

$$l : M \longrightarrow \mathcal{B}$$

where M is a finite set of mail addresses ($M \subset \mathcal{M}$), and $\mathcal{B}$ is the set of all possible behaviors, respectively. We represent the set of all local states functions by $\mathcal{L}$.

A configuration is defined as follows. A restriction on the tags of a configuration (specified in the definition below) is necessary to ensure that there always exist transitions from a given configuration with unprocessed tasks. We wish to avoid any tag conflicts as an actor system evolves.

Definition 5.3 Configurations. *A configuration is a two tuple* (l, T)*, where* l *is a local states function and* T *is a finite set of tasks such that no task has a tag which is the prefix of another tag or mail address.*[1]

Note that the set T in fact represents a function from the a finite set of tags to the cross product of mail addresses and communications. The degenerate case of the prefix relation is equality and thus no two tasks in a configuration may have the same tag.

5.1.2 Requirements for a Transition Relation

What any behavior definition gives us is a map from a finite list of variables to a "behavior." These variables are given specific values whenever any actor is created in the system. An actor's behavior specifies the creation of new tasks and actors as a function of a communication accepted. Newly created actors must have mail addresses that are unique and the different tasks in a system need to be kept distinct.

A global scheme for assigning mail addresses to newly created actors is not a faithful representation of the concurrency inherent in an actor system although such a scheme would provide a simple mechanism for generating new mail addresses in much the same way as the semantics of block declarations in Pascal provides for the creation of new variables [de Bakker 80]. We will instead provide a distributed scheme for generating mail addresses.

One can maintain the uniqueness of tasks by providing distinct tags for each and every task in an actor system. In fact, one purpose of mail

[1] The prefix relation is defined using the usual definition for strings.

addresses is quite similar to that of tags: mail addresses provide a way of differentiating between identically behaving actors. Mail addresses also specify a network topology on actors by allowing one to define a *directed graph* on them (the nodes in such a graph denote the actors). We will use the unique tags of a task to define more unique tags and mail addresses for the new tasks and actors created. Having defined a scheme which guarantees the uniqueness of tags and mail addresses, we can transform the instantiations of the behavior definition into a transition relation from each actor and task to a system of actors and tasks. This transition relation can be extended meaningfully to a system of actors and tasks as long as mail addresses and tags can be generated in a distributed fashion and maintain their uniqueness as the system evolves.

5.2 Initial Configurations

Our goal is to map actor programs to the initial configurations they define. To do so we have to specify how the meaning of the various constructs in an actor program. We confine our consideration to minimal actor languages such as the kernel of SAL and *Act* defined in Section 3.2. Since all the extended constructs are definable in such minimal languages, and since the kernel is much simpler than any expressive extension, such a restricted focus is not only pragmatically desirable but theoretically sufficient .

5.2.1 Formalizing Actor Behaviors

The behavior of an actor was described informally in Section 2.1.3. In a nutshell, we can represent the behavior of an actor as a function from the possible incoming communications to a 3-tuple of new tasks, new actors, and the replacement behavior for the actor. We give a domain for actors below. Since the given domain of actor behaviors is recursive, it is not immediately obvious that the behavior of an actor is well-defined: We can deduce from a simple cardinality argument (following Cantor) that not all functions of the form in definition 5.5 will be meaningful.

There are two ways to resolve the domain problem for actors. The first solution is to use Scott's theory of *reflexive domains* [Scott 72] to map actor behaviors into an abstract, mathematically well-defined space of functions. Applying Scott's theory each actor program denotes a value in the specified abstract space. Such valuations, however, may or may not suggest a means of implementing an actor language. In fact, one can show that computation paths defined using the transition relation specify *information system* as defined in [Scott 82].

In the denotational semantics of sequential programs, a major advantage of the fixed-point approach has been the ability to abstract away from the operational details of the particular transitions representing the intermediate steps in the computation. The sequential composition of functions representing the meaning of programs corresponds nicely to the meaning of the sequential composition of programs themselves. This also implies that the meaning (value) of a program is defined in terms of the meaning of its subcomponents [Stoy 77]. Furthermore, since sequential composition is the only operator usually considered in the case of deterministic, sequential programs, the fixed-point method is fully *extensional* [de Bakker 80].

Unfortunately, fixed point theory has not been as successful in providing extensional valuations of concurrent programs. The problem arises because of the requirements of parallel compositionality: Specifically, the history of a computation is not as easily ignored. We will return to this topic in Chapter 7.

What we propose to do in this chapter is to provide a functional form for the behavior of an actor in a given program. Specifying the meaning of a program in these terms does not abstract all the operational details related to the execution of the code. These functions will in turn be used to define the initial configuration and the transitions between configurations. The representations are entirely *intentional* in character and thus provide constructive intuitions about the nature of computation in actor systems.

Note that the semantics of actor programs developed in this section is *denotational* because the meaning of a program is built from the meaning of its constituent parts. We begin by defining actors and their behaviors.

Definition 5.4 Actors. *The set of all possible actors, $\mathcal{A}$, is given by*

$$\mathcal{A} = \mathcal{M} \times \mathcal{B}$$

where $\mathcal{M}$ is the set of all possible mail addresses (as above), and $\mathcal{B}$ is the set of all possible behaviors.

The tag of the task processed by an actor α is used to define new tags for the tasks, and new mail addresses for the actors, that are created by α in processing the task. Notice that there are only a finite number of tags and mail addresses possible. A recursive domain for actor behaviors is given below.

Definition 5.5 Behaviors. *The behavior of an actor, with the mail address m, is an element of $\mathcal{B}$, where*

$$\mathcal{B} = (\ \mathcal{I} \times \{m\} \times \mathcal{K} \longrightarrow F_s(\mathcal{T}) \times F_s(\mathcal{A}) \times \mathcal{A}\ )$$

where $F_s(\mathcal{T})$ is the set of all finite subsets of $\mathcal{T}$ and $F_s(\mathcal{A})$ is the set of finite subsets of $\mathcal{A}$. Furthermore, let β be a behavior for an actor at mail address m, and t be the tag and k be the communication of the task processed, such that $\beta(k) = (T, A, \gamma)$, where

$$T = \{\tau_1, \cdots, \tau_n\}$$

$$A = \{\alpha_1, \cdots, \alpha_{n'}\}$$

then the following conditions hold:

1. *The tag t of the task processed is a prefix of all the tags of the tasks created:*

$$\forall i\, (1 \leq i \leq n \Rightarrow \exists m_i \in \mathcal{M}\ \exists k_i \in \mathcal{K}\ \exists t'_i \in \mathcal{I}\ (\tau_i = (t.t'_i, m_i, k_i)))$$

2. *The tag t of the task processed is a prefix of all the mail addresses of the new actors created:*

$$\forall i\, (1 \leq i \leq n' \ \Rightarrow\ \exists \beta_i \in \mathcal{B}\ \exists t'_i \in \mathcal{I}\ (\alpha_i = (t.t'_i, \beta_i)))$$

3. Let I be the set of tags of newly created tasks and M be the set of mail addresses of newly created actors. Then no element of $I \cup M$ is the prefix of any other element of the same set.

4. There is always replacement behavior.

$$\exists \beta' \in \mathcal{B} \; (\gamma = (m, \beta'))$$

The example below is for illustrative purposes. The meaning developed in §5.2.2 will allow us to derive from the code the functional form given.

Example 5.2.1 Recursive Factorial. The recursive factorial discussed in section 2 is an example of an unserialized actor. The code for such an actor is given in Section 3.3. The behavior of a recursive factorial actor at the mail address m, $(m\,,\,\varphi)$, can be described as follows:

$$\varphi(t, m, [k_1, k_2]) \; = \; \begin{cases} \langle \{(t.1, k_2, [1])\}\,,\, \emptyset\,,\, (m, \varphi)\rangle & \mathit{if} \;\; k_1 = 0 \\ \langle \{(t.1, m, [k_1 - 1, t.2])\}\,,\, \{(t.2, \psi^{k_1}_{k_2})\}\,,\, (m, \varphi)\rangle & \mathit{otherwise} \end{cases}$$

where m is the mail address of the factorial actor, t is the tag of the task processed. The behavior of the newly created *customer* can be described as

$$\psi^{k_1}_{k_2}(t', t.2, [n]) \;=\; \langle \{(t'.1, k_2, [n * k_1])\}\,,\, \emptyset\,,\, (t.2, \beta_\perp)\rangle$$

where $t.2$ is the mail address of the newly created actor, and t' is the tag of the task it processes. $\beta_\perp$ is *bottom-behavior*, which is equivalent to an infinite sink. Note that it can be shown in any actor system that this newly created actor will receive at most one communication, thus the behavior of its replacement is actually irrelevant.

5.2.2 The Meaning of Behavior Definitions

Recall that an actor machine embodies the current behavior of an actor. Conceptually, an actor machine is replaced with another, perhaps identical, actor machine each time a communication is accepted by an actor. The behavior of an actor machine is quite simple: it involves no iteration, recursion, synchronization, or state change. The behavior is simply a function of the incoming communication and involves sending more communications to specified targets, creating new actors, and specifying a replacement actor machine.[2] We will use the syntactic default in an actor program that whenever there is no *become* command in the code of an actor, then the replacement behavior is simply an identically behaving actor. One can now safely assert that all actors definable in an actor language like SAL specify a replacement behavior. Alternately, we could have decided that a behavior definition which did not provide a replacement in some case was simply meaningless.

In this section, we closely follow the relevant notation and terminology from [de Bakker 80]. Each actor program consists of a finite number of behavior definitions which will form templates for all the behaviors of actors that may be created in the course of program execution. We will define the meaning of a behavior definition as a map from:

- The mail address, *self*, of the actor whose behavior has been defined using the template; and

[2]The rest of this section is a technical justification for a well formed interpretation of actor behaviors and may be skipped without loss of continuity.

- The variables in the acquaintance list of the behavior definition.

And a map into a function mapping a task with target *self* into a three tuple consisting of:

- A set of tasks;
- A set of three tuples consisting of a mail address, a behavior definition, and a list of values; and,
- A three tuples consisting of the mail address *self*, a behavior definition, and a list of values.

We carry out the construction formally. We first define the syntax for the class of *primitive* expressions. There are three kinds of primitive expressions: integer, boolean and mail address expressions. These expressions will occur in different commands. The class *Icon* typically corresponds to identifiers such as $3, 4, -1, \ldots$, while the class *Ivar* corresponds to the identifiers used for integers in a program. Note that there is no class of mail address constants in the expressions of our language because the programmer has no direct access to mail addresses. The primitive expressions given below are purely syntactic objects which will be mapped into mathematical objects by a valuation function.

Definition 5.6 Syntax of Primitive Expressions.

1. *Let* Ivar*, with typical elements* $x, y, \ldots$*, be a given subset of the class of identifiers, and* Icon *be a given set of symbols with typical elements*

$n, \ldots$. The class Iexp, with typical elements $s, \ldots$, is defined by

$$s ::= x \mid n \mid s_1 + s_2 \mid \cdots$$

(Expressions such as $s_1 - s_2$ may be added.)

2. *Let Mvar, with typical elements $a, \ldots$, be a given subset of the class of identifiers, E be an element of* Dvar *(defined later) and $e_1, \ldots, e_i$ be arbitrary expressions, then the class Mexp, with typical elements $h, \ldots$, is defined by*

$$h ::= a \mid \mathsf{new}\ E(e_1, \ldots, e_i)$$

3. *Let Bvar, with typical elements $b, \ldots$, be a given subset of the class of identifiers, and Bcon be the set of symbols $\{\underline{\texttt{true}}, \underline{\texttt{false}}\}$. The class Bexp, with typical elements $b, \ldots$, is defined by*

$$b ::= \underline{\texttt{true}} \mid \underline{\texttt{false}} \mid s_1 = s_2 \mid h_1 = h_2 \mid \neg b \mid \cdots$$

We now assume the existence of three classes of mathematical objects: namely, a class of integers, V, a class of mail addresses, M, and a class of truth values, $W = \{tt, ff\}$. The integers and the truth values have the usual operations associated with them, such as addition for integers. We assume that the concatenation operator works for the mathematical objects called mail addresses since the class of mail addresses will be identical to the class of tags and the latter will be suffixed to define new mail addresses.

Let the set of primitive variables, Pvar, be the separated sum of integer, boolean, and mail address variables.[3] Similarly, let P be the set of primitive values representing the separated sum of the integers, the truth values and the mail addresses. A *local environment* is defined as an element of:

$$\Sigma : Pvar \rightarrow Pval$$

There are three semantic functions that need to be defined to give a meaning to the primitive expressions. Given a local environment these functions map primitive expressions to primitive values. These functions are:

$$\begin{aligned} V &: Iexp \rightarrow (\Sigma \rightarrow V) \\ W &: Bexp \rightarrow (\Sigma \rightarrow W) \\ M &: Mexp \rightarrow (\Sigma \rightarrow M) \end{aligned}$$

The definitions of the first two functions are by induction on the complexity of the arguments and have nothing to do with actor semantics in particular. We therefore skip them. We will define the meaning function below which will provide the valuation for *new expressions.* Essentially, new expressions evaluate to a new mail address. We will assume a single function π representing the separated sum of above three functions such that π maps each expression into its corresponding value given a particular local environment, σ.

We now give the syntax of commands, and using commands, the syntax of behavior definitions. The syntactic classes defined are called *Cmnd* and

[3] Strictly speaking the set *Bvar* is superfluous since boolean expressions can be defined without it. However, we will assume that all three kinds of variables exist and are distinct.

Bdef. The syntax below is a slightly abbreviated form of the syntax used in SAL. The two noteworthy differences between SAL and the syntax below are as follows. First, we allow let bindings only for *new expressions.* The semantics of let bindings in other cases is quite standard, and in any case not absolutely essential to our actor programs. Second, we use *new expressions*, as opposed to arbitrary expressions, in all *become commands.* The semantic interpretation of *becoming* an arbitrary actor is simply to acquire a *forwarding* behavior to that actor (see §3.2.1). The behavior can thus be expressed as a *new expression* using a predefined forwarding behavior and specifying its acquaintance as the expression. The only reason for these simplifications is brevity.

Definition 5.7 Syntax of Behavior Definitions.

1. *The class Cmnd with typical elements $S, \cdots$, given by*

$$\begin{aligned} S ::= {} & S_1 // S_2 \mid \textsf{if } b \textsf{ then } S_1 \textsf{ else } S_2 \mid \\ & \textsf{send } [e_1, \ldots, e_i] \textsf{ to } a \mid \textsf{become new } E(e_1, \ldots, e_i) \mid \\ & \textsf{let } a_1 = \textsf{new } E_1(e_1, \ldots, e_{i_1}) \textsf{ and } \ldots \\ & \qquad \textsf{and } a_j = \textsf{new } E_j(e_1, \ldots, e_{i_j}) \; \{ \; S \; \} \end{aligned}$$

where the use of the identifiers corresponds to their reserved status above. The identifiers $E, \ldots,$ are used as defined below.

2. *Let Dvar be set of pre-defined symbols. The class Bdef with typical elements $D, \ldots,$ is given by*

$$D \; ::= \; \textsf{def } E(p_1, \ldots, p_i)[p'_1, \cdots, p'_j] \; S \textsf{ enddef}$$

The semantics of the class *Cmnd* is defined below. The semantics maps a given local environment into a 3-tuple representing tasks created, actors created and a replacement actor, respectively. Note that actors are simply denoted by a mail address, an element of *Dvar*, and a list of primitive values which will map into the primitive variables used in the behavior definition using the element of *Dvar*. We also assume that two primitive variables, namely *self* and *curr*, of the class *Mvar* are defined by the local environment. *self* represents the mail address of the actor whose code contains the given command and *curr* represents the tag of the task being currently processed. The meaning function is defined on the complexity of the commands. We will not bother to define a complexity measure for the commands but will simply follow the syntactic definition. The details are trivial. Note that σ represents the local environment and $\sigma[\![a/x]\!]$ represents the environment which is equal to σ except that it has the primitive value a for the primitive variable x. The operation $\uplus$ represents a component-wise union (i.e., the three components are union independently).

The meaning function $\mathcal{F}$ maps each command in a given local environment to a three tuple representing the communications sent, actors created and the replacement actor. The meaning of concurrent commands is the component-wise union of the commands themselves, i.e., the communications sent are the communications sent by each and the actors created are the union of the actors created by executing each of the commands. Recall that there may be only one executable become command in the code of an actor for any given local environment. If the union ends up with more than one replacement actor than it does not define an actor behavior. The main point of interest in concurrent composition is the suffixing of the current tags. This mechanism ensures that the new actors and tasks created by the actor will satisfy the prefix condition in definition 5.5. Assume that *curr* is initially bound to t on the left hand side of all the equations given below.

$$\mathcal{F}(S_1//S_2)(\sigma[\![t/curr]\!]) = \mathcal{F}(S_1)(\sigma[\![t.1/curr]\!]) \uplus \mathcal{F}(S_2)(\sigma[\![t.2/curr]\!])$$

The meaning of the *conditional command* and the *send command* is straightforward. The *become command* specifies the replacement behavior by specifying an identifier which will denote a behavior definition and a list of values which will partially determine the local environment in which the command in the definition is executed.

$$\mathcal{F}\ (\textsf{if } b \textsf{ then } S_1 \textsf{ else } S_2)\ (\sigma) = \begin{cases} \mathcal{F}(S_1)(\sigma) & if\,\pi(b) = \mathbf{tt} \\ \mathcal{F}(S_2)(\sigma) & otherwise \end{cases}$$

$$\mathcal{F}(\textsf{ send } [e_1, \ldots, e_i] \textsf{ to } a\ )(\sigma[\![t/curr]\!]) = \\ \langle\{t.1, \pi(a)(\sigma), [\pi(e_1)(\sigma), \ldots, \pi(e_i)(\sigma)]\}, \emptyset, \emptyset\rangle$$

$$\mathcal{F}(\textsf{become new } E(e_1, \ldots, e_i))(\sigma[\![m/self]\!]) = \\ \langle\emptyset, \emptyset, \{(m, E(\pi(e_1)(\sigma), \ldots, \pi(e_i)(\sigma))\}\rangle$$

The creation of new actors is accomplished by *new expressions* and *let bindings*. We have to specify the new mail addresses for all concurrently created actors which may know each others mail address. The command in the scope of the bindings is also executed in an local environment where all the identifiers for the actors are bound to the mail addresses of the newly created actors.

$$\begin{aligned}&\mathcal{F}(\mathsf{let}\ a_1 = \mathsf{new}\ E_1(e_1,\ldots,e_{i_1})\ \mathsf{and}\ \ldots\ \mathsf{and}\\ &\quad a_j = \mathsf{new}\ E_j(e_1,\ldots,e_{i_j})\ \{S\})(\sigma[\![t/curr]\!]) = \quad \mathcal{F}(S)(\sigma') \uplus\\ &\quad \langle\emptyset, \{\alpha_n : \forall 1 \leq n \leq j(\alpha_n = (t.n, E_n(\pi(e_1)(\sigma'),\ldots,\pi(e_{i_n})(\sigma'))\},\emptyset\rangle\end{aligned}$$

where $\sigma' = \sigma[\![a_1/t.1,\ldots,a_j/t.j]\!]$

Now the meaning of a behavior definition is simply to modify the *program environment* by mapping each *Dvar* into the meaning of the command. We skip the (simple) proof that a behavior definition defines behaviors that satisfy the requirements of definition 5.5. The tag and mail address generation schemes we used were intended to satisfy these requirements. The only other constraint of interest is that there be at most one executable become command. The behavior definition is simply not well-defined if its meaning violates this constraint.[4]

5.2.3 Mapping Actor Programs

The basic syntax of a SAL program consists of *behavior definitions* and commands. The commands are used to create actors and to send them communications.[5] Now a program environment associates the identifiers in *Dvar* with the meaning of commands for each behavior definition in the

[4] In an implementation, we would generate an error message.

[5] We are ignoring for the present the receptionist and external actor declarations; although such declarations are useful for imposing a modular structure on the programs, they do not directly affect the transitions internal to the system.

program. All other members of *Dvar* are undefined and may not be used in the commands of a syntactically correct program. The program itself is a single command (recall that concurrent composition of commands is a command) and its meaning is given using the function $\mathcal{F}$ defined above with the local environment as the program environment. The technique used here is similar to that in used in [de Bakker 80] where procedure variables are defined in the denotational semantics of recursion. The syntax of a program can be given as follows:

$$P ::= D_1 \ldots D_n \; S$$

where the D_i's represent behavior definitions and S represents a command (which may, of course, be the concurrent composition of other commands). The variable *curr* is initially bound to 1.

Note that none of the top level commands can be a *become* command because the commands are not being executed by a given actor. Thus an actor program is mapped into a two tuple representing the initial configuration. A transition relation tells us how to proceed from a given configuration by, nondeterministically,[6] removing a task from the system and adding the effects of processing that task. The effects of processing a task are given by the behavior of its target, namely the actors and tasks the target creates and the replacement it specifies.

[6] We will return to the issue of the guaranteeing mail delivery in Section 5.3.

5.3 Transitions Between Configurations

In a sequential machine model, the intuition behind transitions is that they specify what actions might occur "next" in a system. However in the context of concurrent systems, there is generally no uniquely identifiable transition representing the "next" action since events occurring far apart may have no unique order to them (as the discussion in §5.2 indicated). Our *epistemological* interpretation of a transition is not that there really is a unique transition which occurs (albeit nondeterministically), but rather that any particular order of transitions depends on the *frame of reference*, or the *view-point*, in which the observations are carried out. This difference in the interpretation is perhaps the most significant difference between a *nondeterministic sequential process* and the model of a truly *concurrent system*: In the nondeterministic sequential process a unique transition in fact occurs, while in a concurrent system, many transition paths, representing different viewpoints, may be consistent representations of the actual evolution.

Our justification for using a transition system is provided by the work of Clinger [81] which showed that one can always define a (non-unique) *global time* to represent the order of events. Events in Clinger's work were assumed to take infinitesimal time and the *combined order* of events was mapped into a linearly ordered set representing a global time. A global time corresponds to events as recorded by some (purely conceptual) observer.

Remark. Transitions, unlike events, may take a specific finite duration and may therefore overlap in time. This is not a problem in actor systems because of the following:

1. All transitions involve only the acceptance of a communication.

2. There is arrival order nondeterminism in the order in which communications sent are accepted and this arbitrary delay subsumes the precise duration of a transition. Specifically:

   (a) When a particular communication is sent because of a transition need not be explicitly modeled: Although a communication may not have been sent before another transition occurs, this possibility is accounted for by the fact that the communication may not cause the "next" transition.

   (b) When the replacement accepts the next communication targeted to the actor is indeterminate: Thus the time it takes to designate the replacement need not be explicitly considered.

   (c) The above reasoning holds for creation of new actors as well.

Global time[7] in any concurrent system is a *retrospective* construct: it may be *reconstructed* (although not as a unique linear order) after the fact by studying the relations on the events in a parallel system. Information about the order of events in a circumscribed system is often useful. In implementations supporting actor systems, such information is useful in delimiting *transactions.* Transactions are defined by the events affecting the reply to a given request (in particular, the events ordered between

[7]By global time, we mean any linear order on the events in the universe.

the request and its corresponding reply). Transactions are useful tools for debugging a system or allocating resources to sponsor activity. The determination of an order of events (the so-called *combined order* as it combines the arrival order with the order of causal activations) in an implementation is achieved by running the actor system in a special mode where each actor records events occurring at that actor and reconstructing the causal activations by following the communications sent.

The possible ways in which a conceptual observer records events, i.e., the behavior of such an observer, corresponds to that of some nondeterministic sequential process. This correspondence is the reason why nondeterminism is used in mathematical models to capture the parallelism. However, the character of the correspondence is *representationalistic*, not *metaphysical.* In particular, the behavior of a parallel system may be represented by many (consistent) nondeterministic sequential processes corresponding to different observers.

5.3.1 Possible Transitions

In this section, we discuss how actor systems may evolve in terms of their descriptions. A transition relation specifies how a configuration may be replaced by another which is the result of processing some task in the former.

Notation. Let *states* and *tasks* be two functions defined on configurations that extract the first and second component of a configuration. Thus the range of *states* is the set of local states functions and the range of *tasks* is the power set of tasks, where the set of tasks may be treated as functions from tags to the target and communication pairs.

The definition for the possible transition relation essentially shows how an interpreter for an actor language would, in theory, work. It thus specifies an operational semantics for an abstract actor language. Note that defining a language in this manner amounts to specifying its semantics by reduction. We will first define the possible transition relation and then show that such transitions do indeed exist for any arbitrary configuration.

Definition 5.8 Possible Transition. *Let c_1 and c_2 be two configurations. c_1 is said to have a possible transition to c_2 by processing a task $\tau = (t, m, k)$, symbolically,*

$$c_1 \xrightarrow{\tau} c_2$$

if $\tau \in tasks(c_1)$, and furthermore, if $states(c_1)(m) = \beta$ where

$$\beta(t, m, k) = \langle T, A, \gamma \rangle$$

and the following hold

$$\begin{aligned} tasks(c_2) &= (tasks(c_1) - \{\tau\}) \cup T \\ states(c_2) &= (states(c_1) - \{(m, \beta)\}) \cup A \cup \{\gamma\} \end{aligned}$$

In order to show that there exists a possible transition from some given configuration, as a result of processing any given task in that configuration, we need to show that a valid configuration can always be specified using the above equations for its *tasks* and *states*. The proof of this proposition uses the conditions on the tags for tasks in a given configuration to assure the functional form for the *tasks* and *states* of the configuration resulting from the transition.

Lemma 5.1 *If c_1 and c_2 are configurations such that $c_1 \xrightarrow{\tau} c_2$ then no task in c_2 has a tag which is the prefix of the tag of any other task in c_2.*

Proof. (By Contradiction) Let t_1 and t_2 be the tags of two tasks τ_1 and τ_2 in the configuration c_2 such that $t_1 = t_2.w$ for some string of integers w separated by periods. We examine the four possible cases of whether each of the tasks belongs to the configuration c_1.

If $\tau_1, \tau_2 \in tasks(c_1)$ then since c_1 is a valid configuration, we immediately have a contradiction. On the other hand, if neither of the two tasks are in c_1, then by Definition 5.5 the the prefix relation is not valid either.

We can therefore assume that one of the tasks belongs to the tasks of c_1 and the other does not. Suppose $\tau_1 \in tasks(c_1)$ and $\tau_2 \notin tasks(c_1)$. Since $\tau_2 \notin tasks(c_1)$, $\tau_2 \in T$, where T is the set of tasks created in the transition. Thus $\exists i\,(t_2 = t.i)$ which together with the hypothesis that $t_1 = t_2.w$ implies that $t_1 = t.i.w$. But since $\tau_1, \tau \in tasks(c_1)$ we have a contradiction to the prefix condition in the tasks of configuration c_1.

The only remaining case is that of $\tau_2 \in tasks(c_1)$ and $\tau_1 \notin tasks(c_1)$. Now $t_1 = t.i = t_2.w$. If w is an empty string then t is a prefix of t_2 and both are elements of $tasks(c_1)$, a contradiction. If $w = i$ then $t = t_2$ and thus $t_2 \notin tasks(c_2)$. But if w is longer than a single number than t is a prefix of t_2 which also contradicts the condition that they are both tasks in c_1. $\dashv$

Lemma 5.2 *The set $states(c_2)$ in the above definition represents a local states function.*

Proof. We need to show that the none of the newly created actors have the same mail addresses as the actors that already existed in c_1. Suppose (m', β') is a newly created actor as a result of processing the task τ. If t is the tag for the task τ then $m' = t.i$ for some nonnegative integer i. Now if there is another actor with the same mail address in the configuration c_2, from lemma 5.2 we know that it can not be a newly created actor. Thus it is in the domain of $states(c_1)$. Then $m' = t.i$ contradicts the assumption that the tags of tasks in c_1 are not prefixes to any mail addresses in c_1. $\dashv$

Lemma 5.3 *The tags of tasks in c_2 are not prefixes to any mail addresses in c_2.*

Proof. Also straightforward (skipped). $\dashv$

The above three lemmas imply the following theorem.

Theorem 5.1 *If c_1 is a configuration and $\tau \in tasks(c_1)$ then there exists a configuration c_2 such that $c_1 \xrightarrow{\tau} c_2$.*

5.3.2 Subsequent Transitions

Of particular interest in actor systems is the fact that all communications sent are subsequently delivered. This *guarantee of delivery* is a particular form of *fairness*, and there are many other forms of fairness, such as fairness over arbitrary predicates. We will not go into the merits of the different forms here but will consider the implications of guaranteeing the delivery of any particular communication even when there is a possible infinite sequence of transitions which does not involve the delivery of a particular communication sent. To deal with this guarantee of mail delivery, it is not sufficient to consider the transition relation we defined in the last section. We will instead develop a second kind of transition relation which we call the *subsequent transition*. The subsequent transition relation was developed in [Agha 84].[8] We first define a possibility relation as the transitive closure of the possible transition and then use it to define subsequent transition.

Suppose the "initial" configuration of an actor system had a factorial actor and two requests with the n_1 and n_2 respectively, where n_1 and n_2 are some nonnegative integers. Since in this configuration, there are two tasks to be processed, there are two possible transitions from it. Thus there two *possible* configurations that can follow "next." Each of these has several possible transitions, and so on. This motivates the definition of a fundamental relation between configurations which can be used to give actors a fixed-point semantics.[9]

Definition 5.9 Possibility Relation. *A configuration c is said to possibly evolve into a configuration c', symbolically, $c \longrightarrow^* c'$, if there exists a sequence of tasks, $t_1, \ldots, t_n$, and a sequence of configurations, $c_0, \ldots, c_n$,*

[8]Milner brought to our attention that the relation we define here is similar to that developed independently in [Costa and Sterling 84] for a *fair Calculus of Communicating Systems* .

[9]Such a domain does not respect fairness.

for some n, a non-negative integer, such that,

$$c = c_0 \xrightarrow{t_1} c_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c_n = c'$$

Remark 1. If $n = 0$ above, we simply mean the identity relation.

One could show, by straight forward induction, that the "initial" configuration, c_{fact}, with the factorial actor possibly goes to one in which a $n!$ communication is sent to the mail address of the customer in the request to evaluate the factorial of n. When the factorial actor is sent two requests, to evaluate the factorials of the nonnegative integers n_1 and n_2, one can make a stronger statement than the one above: Considering that the computation structure is finite, one can show that there is a set of configurations, C that c_{fact} necessarily goes to such that both the factorial of n_1 and n_2 have been evaluated. The configurations in C have the interesting property that no further evolution is possible from them without communications being sent by some external actor. We call such a configuration *quiescent* (cf. termination of a computation).

Consider the following example which requires concurrent processing of two requests. Suppose the factorial actor (as we defined it in Examples 3.2.2 and 5.2.1) was sent two communications, one of which was to evaluate the factorial of -1 and the other was to evaluate the factorial of n, where n is some nonnegative integer. The behavior of the factorial actor implies that it would embark on the equivalent of a non-terminating computation. More precisely it would send itself a communication with $-k$ in response to a communication with $-k-1$, and so on, and therefore it will not possibly evolve to any configuration which is *quiescent*.

Recall that in the actor model the delivery of all communications sent is guaranteed. This implies that despite the continual presence of a communication with a negative number in every configuration this configuration possibly goes to, it must at some point process the task with the request to evaluate the factorial of n.[10] We can express this sort of a result by defining the following relation on sets of configurations.

Definition 5.10 Subsequent Transition Relation. *We say a configuration, c subsequently goes to c' with respect to τ, symbolically, $c \overset{\tau}{\hookrightarrow} c'$, if*

$$(\tau \in \mathit{tasks}(c) \wedge c \longrightarrow^* c' \wedge \tau \notin \mathit{tasks}(c') \wedge$$
$$\neg \exists c'' \, (\tau \notin \mathit{tasks}(c'') \wedge c \longrightarrow^* c'' \wedge c'' \longrightarrow^* c')$$

Basically, the subsequent transition represents the first configuration which does not contain the task in question. If we defined the set of configurations, C, as follows

$$C = \{c' \mid c \overset{t}{\hookrightarrow} c'\}$$

then the guarantee of mail delivery implies that the configuration c must pass through C. We can define a necessity relation based on the subsequent relation but will not digress here to do so. The subsequent transition thus provides a way of defining a fair semantics by derivation for an actor model.

[10] This in turn results in the request to evaluate the factorial of $n - 1$. Thus by induction we can establish that at some point in its life, this factorial actor will (indirectly) activate a communication $[n!]$ to the mail address of the customer in the corresponding communication.

The model is assumed to have these two transition relations as primitives.

Remark. The subsequent relation defines what may be considered locally infinite transitions. This is due to the nature of nondeterminism in the actor model. The relation captures the unbounded nondeterminism inherent in the actor paradigm. For a discussion of this phenomenon, see [Clinger 81]. Some authors have found unbounded nondeterminism to be rather distressing. In particular, it has been claimed that unbounded nondeterminism could never occur in a real system [Dijkstra 77]. Actually unbounded nondeterminism is ubiquitous due to the quantum physical nature of our universe. For example, it is found in meta-stables states in VLSI [Mead and Conway 80].

Chapter 6

Concurrency Issues

In this chapter, we discuss how the actor model deals with some of the common problems in the theory of concurrent systems. The first section discusses the implications of the actor model for divergence, deadlock and mutual exclusion. The problem of divergence is severely contained by the guarantee of delivery of communications. Deadlock, in a strict syntactic sense, can not exist in an actor system. In a higher level semantic sense of the term, deadlock can occur in a system of actors. Fortunately, even in the case of a semantic deadlock, the structure of actor systems implies that the "run-time" detection of deadlock, and hence its removal, is quite feasible.

In the second section, we discuss issues related to functionality in a system and the imposition of a merge structure on communications. With respect to functionality, we show that the concept of side-effect free history sensitive functional computation in *streams* is similar in at least one abstract way to the specification of replacement behavior in actors. In both cases, *history-sensitivity* is achieved by similar functional mechanisms. Finally, despite the inherent arrival order nondeterminism, we show the ability to define communication channels in actors which in effect preserve the order of communications between actors.

6.1 Problems in Distributed Computing

There are some problems which are peculiar to distributed systems and cause one to require a great deal of caution in exploiting distributed computing. We will discuss three such problems as they relate to actors: namely, divergence, deadlock, and mutual exclusion. In each instance, we will show how the actor model provides the mechanisms to limit, and perhaps to eliminate these problems.

6.1.1 Divergence

A *divergence* corresponds to what is commonly called an "infinite loop." In some cases, such as a process corresponding to a clock or an operating system, an infinitely extending computation is quite reasonable and *termination* may be incorrect (indeed, aggravating!). At the same time, one may wish to have the ability to stop a clock in order to reset it, or bring down an operating system gracefully in order to modify it [Hoare 77]. Thus we would like to program potentially infinite computations that can nevertheless be affected or terminated.

If one looked at the computation tree defined by the possibility transition then the execution method of an actor program would seem to be modelled as *choice-point nondeterminism* [Clinger 81] or *depth search* [Harel 79]. In this execution scheme, an arbitrary pending communication is nondeterministically accepted by its target causing a transition to the next level in a tree. Using choice-point nondeterminism, it is impossible to guarantee the "termination" of a process which has the potential to extend for an arbitrarily long period of time.

Consider the following simple program. We define the behavior of a stop-watch to be a perpetual loop which can be reset by sending it an appropriate communication (an actor with such behavior may even be useful as a real stop watch, if we had some additional knowledge about the time it took for such an actor to receive and accept the "next" communication it sends itself).

```
stop-watch(n)
    if message = ⟨go⟩
        then  become new stop-watch(n + 1)
              send ⟨go⟩ to self
        else  become new stop-watch(0)
              send [n] to customer
```

Suppose x is created with the behavior *stop-watch* (0). If x is sent a "go" message, then x will embark on a nonterminating computation. If we wish to "reset" x, we can send it another communication, such as [*customer*, "reset"], where *customer* is the mail address of some actor. If and when this message is processed, x will be "reset." Without the guarantee of delivery of communication, however, the "reset" message may never be received by x and there would be no mechanism to (gracefully) reset the *stop-watch.* Since the actor model guarantees delivery of communications, x will at some point be "reset." It will subsequently continue to "tick."

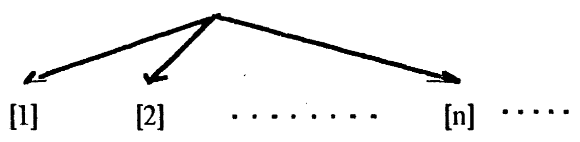

Figure 6.1: *When a* reset *message is processed, one of an infinite number of messages may be sent to the customer. The stop-watch will subsequently continue to tick.*

In the case of the *stop-watch* the potentially perpetual activity affects subsequent behavior. This need not always be the case. A "nonterminating" computation can sometimes be "infinite chatter." Indeed, this is the definition of divergence in [Brookes 83]. We have seen an example of this kind of divergence in the case of our factorial actor when it was sent a message with -1. In a language where the factorial is defined using a looping construct, the factorial could be rendered useless once it accepted a message containing -1. This is because it would embark on a nonterminating computation and therefore never exit the loop in order to accept the next communication. The nontermination of a computation in a language using loops inside a process is a serious problem in the context of a distributed system. The process with an infinite loop may be a shared resource, as would most processes in a network. Since the process is never "done," any other process wishing to communicate with it may not do so and one can have a domino effect on the ability of the system to carry out other computations.

One solution to this problem is to use multiple *activations* of a process. In this framework, each communication to the factorial would activate a process of its own. Activations solve the problem for *unserialized* behavior, as is the case with the factorial. However, when we are interested in a shared object which may actually change its behavior, as is the case in a *stop-watch*, multiple activations are *not* a solution.

The actor model deals with the problem of divergence whether or not the behavior of actors involved is serialized. Divergence, defined as endless chatter, does not affect other activity in the constructive sense that all pending communications are nevertheless eventually processed. The presence of such divergence simply degrades the *performance* of the system.[1] The guarantee of mail delivery also fruitfully interacts with divergence as the term is used in the broader sense of any (potentially) nonterminating computation.

[1] Using resource management techniques, one can terminate computations which continue beyond allocated time. The actor always has a well-defined behavior and would be available for other *transactions* even if some concurrently executing transactions run out of resources.

6.1.2 Deadlock

One of the classic problems in concurrent systems which involve resource sharing is that of *deadlock*. A *deadlock* or *deadly embrace* results in a situation where no further evolution is possible. A classic scenario for a deadlock is as follows. Each process uses a shared resource which it will not yield until another resource it needs is also available. The other resource, however, is similarly locked up by another process. An example of the deadlock problem is the *dining philosophers* problem [Dijkstra 71]. The problem may be described as follows. Five independent philosophers alternately eat and philosophize. They share a common round table on which each of them has a fixed place. In order to eat, each philosopher requires two chopsticks.[2] A philosopher shares the chopstick to his right with the neighbor to the right and like-wise for the chopstick to his left. It is possible for all the philosophers to enter, pick up their right chopsticks and attempt to pick up the left. In this case, none of the philosophers can eat because there are no free chopsticks.

The behavior of a philosopher and that of a chopstick is described as follows:[3]

[2] The usual version is two forks. However, it has never been clear to me why anyone, even a philosopher, would require two forks to eat!

[3] Since we are using expressions, if we were in SAL or *Act*, we would have to specify the behavior in case the reply from the chopstick was not *free*. However, the problem of deadlock has been formulated with no defined behavior in such cases.

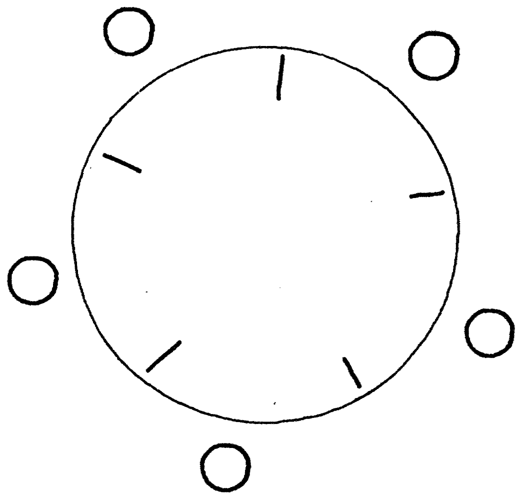

Figure 6.2: *The scenario for the Dining Philosphers problem. Shared resources can lead to deadlock in systems using synchronous communication.*

```
phil ( left-chop , right-chop) [ ]
    let x = call right-chop [ pick ]
    and y = call left-chop [ pick ]
         { if x = free and y = free then ⟨eat⟩ ... }

chopstick(state) [k]
    if k = pick and if state = "free"
            then  reply [free]
                  become new chopstick (busy)
    ...
```

A solution to this problem is to define a *waiter* who acts as a receptionist to the dining area: The waiter can make sure that no more than four philosophers attempt to eat simultaneously. In this case, at least one philosopher will be able to pick up two chopsticks and proceed to eat. Subsequently, this philosopher would relinquish his chopstick allowing another philosopher to eat [Brookes 83].

The "waiter" solution is a particular example of the strategy of access to a shared resource in order to avoid the possibility of deadlock. The difficulty with this solution is that the mechanisms for avoiding deadlock have to be tailored using advance knowledge about how the system might deadlock. Furthermore, the *waiter* is a bottleneck which may drastically reduce the throughput in a large system. However, this is the only sort of solution in systems relying on synchronously communicating sequential processes. In fact the philosopher who picks up his right chopstick can not communicate with his left chopstick as the left chopstick is "busy" with the philosopher to that chopstick's left. Furthermore, even if a philosopher, $phil_1$, knew that he shared his left chopstick with another philosopher, say $phil_2$, unless $phil_2$ was already ready to communicate with $phil_1$, the latter could not send a message to the former. In such a system, not only is there a deadlock, but there is no mechanism for detecting one. In fact in languages using synchronous communication, deadlock has been defined as a condition where no process is capable of communicating with another [Brookes 83].

Two areas of Computer Science where the problem of deadlock arises in practice are operating systems and database systems. In operating systems, deadlock avoidance protocols are often used. However, in database systems it has been found that deadlock avoidance is unrealistic [Date 83]. The reasons why deadlock avoidance is not feasible in concurrent databases can be summarized as follows:

- The set of *lockable objects* (cf. chopsticks in the dining philosophers example) is very large— possibly in the millions.
- The set of lockable objects varies dynamically as new objects are continually created.
- The particular objects needed for a *transaction* (cf. "eating" in our example) must be determined dynamically; i.e., the objects can be known only as the transaction proceeds.

The above considerations are equally applicable to the large-scale concurrency inherent in actor systems. The actor actor model addresses this problem in two ways. First, there is no *syntactic* (or low-level) deadlock possible in any actor system, in the sense of it being impossible to communicate (as in the Brookes' definition above). The chopstick, even when "in use," must designate a replacement and that replacement can *reply* to the philosopher's query. What sort of information is contained in the reply, and what the philosopher chooses to do with the reply depends on the program. If each philosopher is programmed to simply keep trying to use the chopstick then, indeed, in a *semantic* sense, the system can be deadlocked. However, notice that one can specify the behavior of the chopstick so that the replacement replies with information about who is using it even while it is "busy." Thus, $phil_1$ can query $phil_2$ about $phil_2$'s use of the chopstick shared between them. In the end, it would be apparent to the inquisitive philosopher that he was waiting for himself— which is the condition for deadlock.

The most involved aspect of a deadlock is *detecting* it. Since in the actor model, every actor must specify a replacement, and mail addresses may be communicated, it is possible to detect deadlock. An actor is free and able to figure out a deadlock situation by querying other actors as to their local states. Thus an actor need not be prescient about the behavior of another actor. For example, the ability to communicate mail addresses means that a philosopher need not know in advance which other philosopher or philosophers may be using the chopstick of interest. These philosophers, while they may be "busy" eating or looking for a chopstick, nevertheless are in turn guaranteed to accept communications sent to them, and thus a system can continue to evolve.

Our solution is in line with that proposed for concurrent database systems where "wait-for" graphs are constructed and any cycles detected in the graphs are removed [King and Collmeyer 73]. We would carry out the process of breaking the deadlock in a completely distributed fashion. A concern about deadlock detection is the cost of removing deadlocks. Experience with concurrent databases suggests that deadlocks in large systems occur very infrequently [Gray 1980]. The cost of removing deadlocks is thus likely to be much lower than the cost of any attempt to avoid them.

A system of actors is best thought of as a community [Hewitt and de Jong 83]. Message-passing viewed in this manner provides a foundation for reasoning in open, evolving systems. Deadlock detection is one particular application of using message-passing for reasoning in an actor system: Any actor programmed to be sufficiently clever can figure out why the resource it needs is unavailable, and without remedial action, about to stay that way. To solve this sort of a problem, *negotiation* between independent agents becomes important. In open and evolving systems, new situations will arise and thus the importance of this kind of flexibility is enormous.

Another consequence of "reasoning" actors is that systems can be easily programmed to *learn:* A philosopher may become one that has learned to query some particular philosopher who is a frequent user of the chopstick it needs instead of first querying the chopstick. Or the actor may become one which avoids eating at certain times by first querying a clock.

6.1.3 Mutual Exclusion

The *mutual exclusion* problem arises when two processes should never simultaneously access a shared resource. An actor represents *total containment*, and can be "accessed" only by sending it a communication. Furthermore, an actor accepts only a single communication and specifies a replacement which will accept the next communication in its mail queue. The actor may specify a replacement which simply buffers the communications received until the resource is free. We have seen an example of this strategy with *insensitive actors.* Although, a single receptionist may control access to a resource, the resource itself can still be modelled as a system of actors so that there may be concurrency in the use of the resource. There can also be multiple receptionists in a system, when this is appropriate. The mutual exclusion problem, it can be safely asserted, is not really a problem for actors.

6.2 Graphic Representations

In this section, we deal with some of the graphical aspects of the communication patterns. First, we discuss the analogy between the ability to send oneself communications in dataflow and the replacement model in actors. We establish the functionality in the behavior of actors by defining nodes in the spirit of dataflow graphs to illustrate the similarity. Second, we treat the problem of communication channels and the ability, in actors, to constrain the effects of universal nondeterministic merges without defining any new construct.

6.2.1 Streams

A *stream* is a sequence of values passing through a graph link in the course of a dataflow computation [Weng 75]. Streams were introduced for 'side-effect free history-sensitive computation'. In this section, we show by analogy to streams, that actors are also side-effect free in the same sense of the term. To see the equivalence, consider each node as containing a single behavior definition which is equivalent to all the behavior definitions which may be used by the replacements. The fact that there may be a sequence of definitions in a SAL program is a matter expressive convenience. Actually, having more than one behavior definition does not really change anything. The identifier used in a new expression is simply a selector of which behavior definition to use and this fact can be communicated just as well. There are only a finite number of definitions, and the identifier corresponding to each one is simply a selector. A single behavior definition which receives an identifier and branches on it to the code corresponding to the behavior would have an equivalent effect. The *become* command in the program is equivalent to sending oneself a communication with the values of acquaintances including the identifier corresponding to the definition to be used in order to determine the replacement behavior.

There is an apparent difference between actors and nodes in dataflow graphs; in dataflow the values "output" form a single stream. So the correspondence can be visualized more closely using the picture Fig. 6.3 which uses appropriate filters on the stream to separate the message intended for the actor itself and that intended for "output."

Of course actors, unlike the elements of dataflow, do more than pass streams— they may change their acquaintances and they may create other actors. Furthermore, actors themselves are <u>not</u> sequential in character and the replacement is concurrently executed. One consequence of this is the ability to use recursive control structrures which can not be used in static dataflow. One variation of the dataflow model allows for fully re-entrant code by tagging the "tokens" (messages) [Gurd et al 85]. This, however, results in forcing the computation through a possible bottleneck instead of

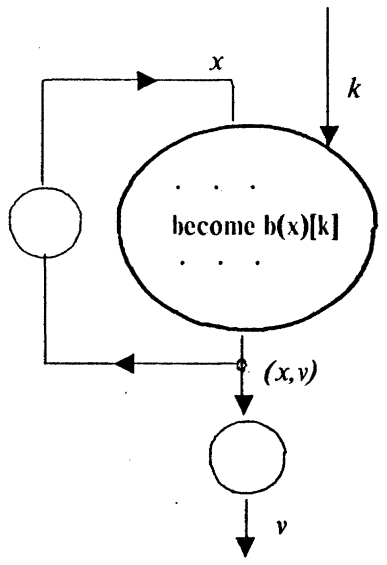

Figure 6.3: *The replacement of an actor can be computed using streams which feed the value of the requisite identifiers for the new behavior. Actors can separate the values needed for replacement from those "output."*

distributing it as is conceptually feasible. The cause of this limitation is the static nature of inter-node topology. Although the actor model allows for dynamic creation, the behavior of an actor is nevertheless functionally determined.

6.2.2 Message Channels

Many systems preserve the order of messages between processes. A stream in dataflow is defined as a sequence of values, and thus by definition is ordered. This creates the interesting problem in dataflow when the order of input from two sources can not be pre-determined. A special element for non-deterministic *merge* has to be assumed and such an element can not be defined in terms of the other constructs in the dataflow model.

The preservation of the order of messages between processes is sometimes simply a function of the hardware configuration. For example, in point-to-point communication between two processors the message channel preserves the order of communications. Sometimes this property can be usefully exploited in computations. An example of this kind of use is found in [Seitz 85] which describes as architecture based on 64 processors, called the *Cosmic Cube*. In Seitz's system, multiple processes may reside on a single processor but processes are never migrated. The processes are asynchronous and use message-passing to interact. However, unlike actors the messages are sent along fixed channels so that (coupled with the lack of migration of processes) the order in which messages are sent by a process A to a process B is the same order in which B receives those messages (although other messages may be inter-leaved).

There are two problems with the strong hardware-based order preservation of message. First, the failure of a single processor would be disastrous since one couldn't re-route a message and necessarily preserve its order in transmission with respect to other messages already sent. Secondly, this scheme creates difficulties in load balancing which requires variable routing of messages and migration of processes. It is for these reasons that the actor model assumes nondeterminism in the relation between the order in

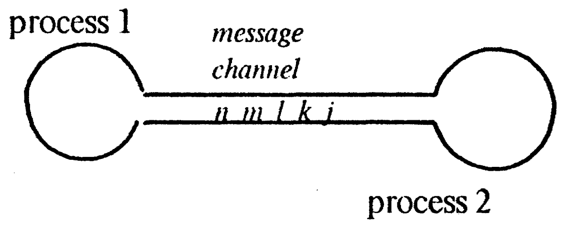

Figure 6.4: *A communication channel preserves the order of communications between two processes. Such channels can be readily defined in actor systems.*

which communications are sent and the order in which they are received. Such nondeterminism is termed *arrival order nondeterminism.*

It is nevertheless possible to define actors which preserve the order in which they, in effect, process communications from each other. Suppose we wished that an actor f "processed" communications from an actor g in the same order as they were sent by g. What the actor g needs to do is tag each message it sends to f with a *reference number* and increment that number each time. The actor f in turn remembers the number of messages it has so far processed from g. If it has processed two, and *message number 4* from g arrives next, f simply buffers that communication until it has accepted *message number 3* from g. Since the delivery of communications is guaranteed, the communication enclosing *message number 3* will also arrive. Subsequently, the actor f will check its buffer for *message number 4* and proceed to process the same. The details can be easily written out in SAL. We have shown that it is not necessary to add any new constructs in order to define order-preserving communication channels in an actor language.

The scheme we use to show the definability of channels is similar to that used in Computer Network Architectures where *sequence numbers* are used in *packet switched networks* to carry out sequence checking [Meijer and Peeters 83]. However, unlike network architectures, we do not make ubiquitous use of virtual network channels because doing so would generally have the effect of slowing the speed with which parallel computations were actually carried out.

Chapter 7

Abstraction And Compositionality

The ability to write and debug large software systems is strongly dependent upon how successfully the system can be partitioned into *independent* modules. Two modules are independent if we can ignore the internal details and treat them as black-boxes with certain input-output behavior. Concurrency involves a nondeterministic interleaving of events; one consequence of such interleaving is that when systems are composed, events in one system are interleaved with the events of the other. Unfortunately, the behavior of the composed system is not necessarily deducible from the abstract representations of the behaviors of the modules composed. In this chapter, we address these issues as they relate to concurrency in general and our actor language in particular.

7.1 Abstraction

A classic problem in concurrent systems is the difficulty abstracting away from the operational behavior of a system. Consider a programming language with the assignment command. Let $\mathcal{F}$ be the function which maps commands in this programming language to their meanings. Let $S_1 \equiv x := x + 1$ be a command in this language. If n is any given integer, the state $\sigma[\![n/x]\!]$ stands for a state where n is the value of x. Now the meaning of S_1 can be expressed as:

$$\mathcal{F}(S_1): \quad \sigma[\![n/x]\!] \longrightarrow \sigma[\![n+1/x]\!]$$

Similarly, if $S_2 \equiv x := 2 * x$ then the meaning of S_2 is given by

$$\mathcal{F}(S_2): \quad \sigma[\![n/x]\!] \longrightarrow \sigma[\![2 * n/x]\!]$$

If we were to compose the commands S_1 and S_2 sequentially, then their meaning functions would also be composed sequentially. However, if we are going to compose the two commands concurrently then the situation is not so simple. Suppose that the command S represents the concurrent composition of the S_1 and S_2, i.e., $S \equiv S_1 \parallel S_2$, where $\parallel$ represents concurrent composition. The meaning of S is not obvious: If we started in a state where $x = 2$, then two of the possibilities are that S_1 precedes S_2 or the other way round. In each case, we can deduce the meaning of S by sequentially composing the meanings of S_1 and S_2: Thus after the execution of both commands, x may be 6 or it may be 5. However, it is also possible that the execution overlaps in time. For example, to execute

S_1 we may FETCH the value of x, but before S_1 has finished execution, x may be fetched by S_2. In this case, the "final" state could be $\sigma[\![3/x]\!]$ or $\sigma[\![4/x]\!]$, neither of which is obvious from a composition of the denotational meanings of the two commands.[1]

7.1.1 Atomicity

The solution to this problem is usually given by specifying which commands are *atomic*, i.e., by specifying that the execution of certain commands may not be interrupted [de Bakker 80]. For example, if the assignment command is defined to be "atomic," then one would need to interleave only the meanings of assignment commands.

The problem with the atomicity solution is that it fixes the level of granularity or detail which must be retained by the meaning function. Thus, if the assignment command is atomic, one must retain information about each and every transformation caused by an assignment in a program. This necessarily means that one can not abstract away the operational details to any higher degree. If, for example, one is defining the abstract meaning of an iterative loop, all the transitions involved in the iterative loop must be retained by the meaning function.

The approach to abstraction in actor systems is somewhat different.

[1]It may appear that

$$\mathcal{F}(S_1 \parallel S_2) = \mathcal{M}(S_1;S_2) \cup \mathcal{F}(S_2;S_1) \cup \mathcal{F}(S_1) \cup \mathcal{F}(S_2)$$

but one can construct a slightly more complex example where this relation does not hold either.

The concept of a *receptionist* is defined to limit the interface of a system to the outside. The use of the receptionist concept is illustrated in the context of the assignment example. We define two systems whose behavior differs partly because the receptionist in each is used to attain a different level of abstraction. Consider the first system: x is the receptionist in this system and the behavior of x is as follows:

```
$x(n)$ [$\langle request \rangle$]
    if $\langle request \rangle$ = FETCH then reply [$n$]
    if $\langle request \rangle$ = STORE $i$ then become new $x(i)$
```

This system has the a level of granularity where the behavior of the configuration must be considered in terms of interleaving FETCH and STORE messages. However, in a larger system x may no longer be a receptionist and it may be possible to avoid this level of detail. For example, let r be the receptionist for an actor system and the behavior of r be given as follows:

```
$r(n)$ [$\langle request \rangle$]
    if $\langle request\ is\ to\ assign\ value\ f(x) \rangle$
        then let $a{=}f(n)$ { become new $r(a)$ }
    if $\langle request\ is\ to\ show\ value \rangle$
        then reply [$n$]
```

Note that the nesting of the *become* command inside the let expression creates a sequentiality in the execution (see the discussion about the let construct in §4.3). In this larger configuration, one need no longer consider the FETCH or STORE events. The level of granularity is comparable to the "atomicity" of assignment commands. However, we can define yet larger systems with other receptionists so that these operational details can be ignored as well. We illustrate this concept by means of another example.

7.1.2 Nesting Transactions

Consider a *bank account* which may be accessed through different money machines. Suppose further that this bank account is shared between several users. The behavior for such a bank account may be something like that in Example 3.3. Now one may want that once the account is accessed through a money machine, it should complete the *transactions* with the user at that machine before accepting requests for transactions from other users. The definition of the bank account as given in example 3.3 implies that the bank account processes one request at a time but that it may interleave requests from different "users" and "money machines." To create a system where transactions at each money machine are completed before other transactions are acceptable, we define a larger configuration where the receptionist for the system is some actor called *account-receptionist.* All communications to the account must be sent through this receptionist and the transactions of the account-receptionist consist of several sub-transactions with the users. The behavior of the receptionist may be described as follows:

```
a-free-account
    become ⟨a-busy-account with the current customer⟩
    ⟨process the request⟩
a-busy-account
    if customer ≠ ⟨current customer⟩
         then send ⟨request⟩ to buffer
    if customer = ⟨current customer⟩
         then if ⟨request = release⟩
                then  send ⟨release⟩ to buffer
                      become a-free-account
                else ⟨process the request⟩
```

What the receptionist does is to prevent the interleaving of requests to the account from different users. An analysis of the behavior of this system can thus be done by considering the overall results of transactions from each machine without having to consider <u>all</u> the possible orders in which the requests from different machines may be recieved. We need to consider the possible order in which entire sets of sub-transactions may occur (since the order in which the first request from a user is received is still indeterminate).

One can construct arbitrarily complex systems so that their behavior is increasingly abstract. There is no pre-determined level of "atomicity" for all actor systems. Instead, it is the programmer who determines the degree of abstraction; the concept of receptionists is simply a mechanism to permit greater modularity and hence procedural and data abstraction.

7.2 Compositionality

One of the desirable features about Milner's *Calculus of Communicating Systems* (CCS) is that it models compositionality rather effectively. Milner's notion of composition is based on *mutual experimentation* by two machines: a machine S offers experiments which may combine with experiments of another machine T to yield an "interaction." Both machines, as a result of the interaction, change their local states. Milner's notion of interaction is based on intuitions of how machines may be plugged together physically, a notion that relates very well to synchronous communication.

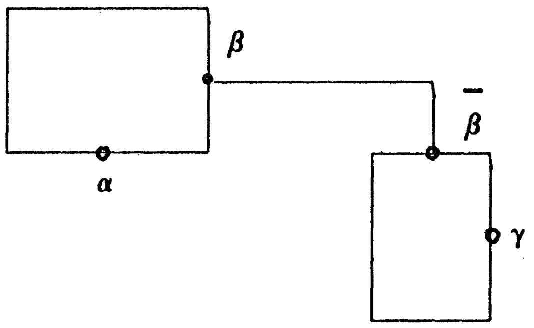

Figure 7.1: *Synchronous composition: In CCS, composition of systems is analogous to plugging machines together. Figure from [Milner 80].*

When an interaction between two machines occurs in Milner's system, one simply links the ports on the two machines. Ports which may be linked are considered *complimentary ports.* One can *hide* a port provided one also hides its complement. Thus, upon composition, one can abstract away from the ports by an operation called *restriction.*

7.2.1 Actors and Ports

The notion of hiding ports using the restriction operator has somewhat different implications in CCS than its intuitive interpretation seems to be when thinking in terms of actors. When a port and its complement have been restricted, its the *interaction* between the two that has been hidden. The port as a result of the interaction, will subsequently unfold its behavior and this behavior will not be hidden. Thus, to use actor jargon, the port may "become" another port which may even have the same label as the one that is hidden. In terms of actors, the restriction operator is equivalent to hiding the acceptance of a single communication; it is not equivalent to hiding all the communications that may be received by the given actor.

A system of actors is best thought of as a *community of agents.* The agents retain their identity even as their behavior changes. Actors have mail addresses which permanently identify them. The "behavior objects" in CCS do not necessarily maintain any "identity" as they interact with the other objects. For example, in CCS, once an agent accepts an input, it may never accept another input on the same port, as it may no longer have the same port as a result of processing the communication. Besides, the ports do not uniquely identify an agent since different agents may use the same "port" name and thus different complimentary ports may be linked. In contrast, communications in actors are sent to a specific unique target actors.

There are other differences between the behavior of an agent in CCS and that of an actor. One of these is that agents in CCS are themselves sequential in character: only *one* experiment may be performed at a time. The justifications Milner provides for this sequentiality are:

1. Tractability of the model; and,

2. The desire to have a "behavior object" represent the system according to an observer capable of only one experiment at a time.

We gave a similar argument about the import of using nondeterminism to model concurrency (§5.2). The fact remains that concurrency includes the potential for overlap in time in both models. There is, however, a fundamental difference between Milner's "behavior objects" and the behavior of an actor: the actor itself is a concurrent agent. The difference is reflected in the language defined by Milner to illustrate CCS and actor languages: in the former, sequentiality is intrinsic; in the latter, it is present only due to causal interactions (§4.2).

7.2.2 Encapsulation in Actors

Any system must have receptionists which can accept information from the "outside," and any system must know of agents that are external to the system. The designation of receptionists and external actors provides for structuring the input-output behavior (or, what in other fields would be called the stimulus-response or sensori-motor aspects) of a system. There are several observations to be made here which are relevant to actor systems:

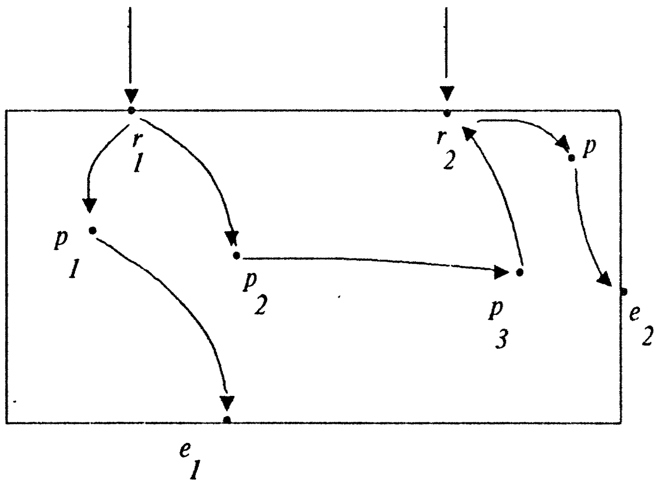

Figure 7.2: *Communications sent by p_3 to r_2 are not observable in an encapsulated system, just as those sent by r_1 to p_1 are "internal."*

- An actor which serves as a receptionist may also be known to other actors within the system. Communications between such "internal" actors and a receptionist will not be observable. Thus it is not so much an actor that is visible or hidden, but rather it is communications between a given *sender-target* pair that are observable when <u>either</u> the sender <u>or</u> the target is external. (See Fig. 7.2.)

- As a system evolves, *new receptionists* may be added and *new external actors* may become known. The mechanism for this change is simply the ability to send messages containing mail addresses.

- One can not arbitrarily restrict receptionists: Once a mail address has been communicated to the outside, it is available for use by external actors. However, if a mail address is unknown to the outside, or becomes unknown, then the actor is no longer a receptionist.

7.2.3 Composition Using Message-Passing

Compositionality in actors is achieved by message-passing. Independent systems are connected by sending some external actors in each module a communication to become *forwarding* actors which simply send their mail to some receptionists in the other module. The justification for the term "become" in the specification of replacement actors is the same as the reason why the external actors and receptionists they forward their mail to are equivalent. We observe the following:

Proposition: If the behavior of an actor x is unserialized, and its behavior is to forward all the communications it accepts to an actor y, then sending a communication to x is equivalent to sending the communication to y.

The proposition is seen to be true because of the arrival order nondeterminism in the actor model. A communication sent to the actor x will be eventually recieved by the actor y. Since the arrival of a communication is always subject to an arbitrarily delay, even if the communication was originally targetted to y, it would have arrived at some indeterminate time at y. Note that the guarantee of delivery is essential in establishing this proposition because otherwise it would be possible for x to recieve the communication and yet y to never receive it.

The rigorous proof of the above proposition would require us to show that given a configuration with a forwarding actor, we can construct an equivalent configuration without the forwarding actor replacing the actor's mail address in the acquaintance lists of all actors and tasks in the configuration defined with the mail address of the actor to which it forwards the communications it receives. The two configurations must be shown to be equivalent in some semantic sense. When an actor, x, acquires the unserialized behavior to forward all communications it receives to an actor y, the actor x is said to *become* y. Using the above proposition as justification we will assume that two such actors are one and the same.

The rules to compose configurations are developed and these may be used to compose systems by composing configurations they may be in. All composition will be done using message-passing, and as a consequence there is no need to assume uniqueness in the configuration at the "time" of composition of a system: The impact of the composition is nondeterministic because of the arrival order nondeterminism in the communications which are forwarded. Since there is arrival order nondeterminism for all messages in actors, no special construction is necessary to the represent the composition of two systems.

7.2.4 Rules for Composition

In this section we develop the constraints that must be satisfied by any scheme which carries out the composition of two systems. We provide the constraints in terms of configurations and assert their realizability by showing a definition of composition which would satisfy the conditions. To compose actor programs, one would map them to the initial configurations they define and compose these configurations using the rules of composition given.

Constraints on Interface Actors

We first define all the applicable rules for constraining the actors that interface with the outside— i.e., set of receptionists and external actors for a composed system.

Let $extern(c_1)$ represent the actors which are external to a configuration c_1, and $recep(c_2)$ represents actors which serve as receptionists in some configuration c_2, then there may be some actor x such that $x \in extern(c_1) \cap recep(c_2)$. It is also possible (but <u>not</u> necessary) that when c_1 and c_2 are composed, such an actor x is no longer a receptionist of the composed system because the only actors x may have been a receptionist for are in the other system composed. In any case, x will <u>not</u> be external to the composed system. Let $c \equiv c_1 \parallel c_2$, where $\parallel$ represents the composition operator. We can assert the following properties about the receptionists and external actors of c:

1. All receptionists in the composed system must be receptionists in one of the two configurations:

$$recep(c) \subset recep(c_1) \cup recep(c_2)$$

2. The only actors which may no longer be the receptionists are actors that are external to one of the configurations composed:

$$(recep(c_1) \cup recep(c_2)) - recep(c) \subset extern(c_1) \cup extern(c_2)$$

3. All external actors in a composed configuration must be external to one of the two configurations:

$$extern(c) \subset extern(c_1) \cup extern(c_2)$$

4. The only actors which may no longer be external are actors that are receptionists for one of the configurations composed:

$$(extern(c_1) \cup extern(c_2)) - extern(c) \subset recep(c_1) \cup recep(c_2)$$

Since we wish to have the identifiers in an actor program (and correspondingly mail addresses in a configuration) be local to the module (or to the configuration), we have to provide a means of "relabeling" the same so as to link receptionists and external actors. Thus when two program modules are composed, we may have a declaration of the form:

$$\mathsf{let}\ \ id_1 = id_2\ \ \mathsf{and}\ \ id_3 = id_4\ \ \ldots$$

where id_1 is the identifier for an external actor in the first module, and id_2 is an identifier for a receptionist in the second, or vice-versa. Similarly for id_3 and id_4, and so on. The intended interpretation of the above declaration is that in order to compose two modules, we simply send an appropriate communication the external actors in each of the modules telling them which receptionist in the other module they should become.

One can not necessarily deduce the receptionists of a composed system from the receptionists of its constituents: Some receptionists may have been so designated only because they were supposed to represent *external* actors in the 'other' module. Thus a new receptionist declaration may be given for a composed system, provided that such a declaration satisfies properties 1 and 2 above.

Formalizing Composition

We now turn to developing a detailed definition for composing two configurations. To be precise, assume a configuration is a four tuple with the functions *states, tasks, recep,* and *extern* extracting each component. (We are expanding the definition of a configuration used in chapter 5 which was concerned more specifically with the internal evolution of an actor system and thus took into account only the first two components.) The *population* of a configuration c, $pop(c)$, consists of mail addresses that are in c but are not elements of $extern(c)$. Suppose c_1 and c_2 are two configurations. To compose c_1 and c_2, we need to specify the new receptionists and external actors. Notice that if c_1 and c_2 are arbitrary configurations and we assume mail addresses are local to a configuration (recall that mail addresses are merely ways of labeling actors to specify topological properties of a system), then there is no guarantee that $pop(c_1) \cap pop(c_2) = \emptyset$. Similarly, if $tags(c)$ is the set of tags used in the $tasks(c)$, then it is possible that

$tags(c_1) \cap tags(c_2) \neq \emptyset$.

In fact, even if the populations and tags of two configurations are disjoint, the *states* and the *tasks* can not be simply combined using the union operation. To see why, recall the prefix condition in the definition of a configuration (Definition 5.3) and its use in Theorem 5.1: The condition states that no tag be the prefix of any other tag or mail address in a configuration. This property is necessary to maintaining the uniqueness of all tags and mail addresses of tasks created.

Tags and mail addresses have no reality of their own. They are merely labels we define to keep a track of computation in an actor system. So we will provide a map to new tags and mail addresses in a composed system so that the new tags maintain the structure implied by the original tags and at the same time satisfy the requisite constraints. Providing a map to carry out the composition has no intrinsic value but simply demonstrates the ability to carry out composition.

Definition 7.1 Composition. *Suppose that c, c_1 and c_2 are configurations such that $c = c_1 \|_{D,R} c_2$, where D is a declaration equating external actors and receptionists, and R is a receptionist declaration satisfying the constraints given above. Let the declarations in D be equivalences of the form $i.e \approx j.r$ where $i, j \in \{1,2\}$, $e \in extern(c_i)$ and $r \in recep(c_j)$. Then the following conditions hold:*

1. *The tags and mail address are simply prefixed by the configuration they came from. Thus,*

$$tasks(c) = \{(i.t, i.m, k') \mid (t, m, k) \in tasks(c_i) \wedge k' = k[\![i.t/t, \ldots]\!]\}$$

2. *The states of all actors not in the declaration D are unchanged except of the transformation on the mail addresses. Let* $\underline{forwarding}(x)$ *represent the behavior of an actor which sends all communications it accepts or has buffered on to x, then*

$$states(c)(i.m) = \begin{cases} \underline{forwarding}(j.r) & if\ i.m \approx j.r\ in\ D \\ b & otherwise\ given\ (m, b) \in c_i \end{cases}$$

3. *The external actors are those who have not been declared to be equivalent to some receptionist in the composed system.*

$$\begin{aligned} extern(c) = \ & (extern(c_1) - \{x \mid \exists r \in recep(c_2)(1.x = 2.r \in D)\} \ \cup \\ & (extern(c_2) - \{x \mid \exists r \in recep(c_2)(2.x = 1.r \in D)\}) \end{aligned}$$

4. *The receptionists of c are given by the declaration R.*

Note that our definition can be easily extended to composition of an arbitrary number of configurations. Parallel composition should of course be commutative and associative. In our definition, the configurations themselves would be different depending on the order of composition. However, there is a strong equivalence relation between them, namely a *direct relabeling* equivalence. Since there are only a finite number of tags and mail addresses the problem of determining the equivalence of any two configurations is decidable.

To compose already existing systems, we need to compose all the configurations the systems may be in. If we use the $c_1 + c_2$ to represent the fact that a system may be in configuration c_1 or in configuration c_2 then:

$$(c_1 + c_2) \parallel (c_3 + c_4) = (c_1 \parallel c_3) + (c_1 \parallel c_4) + (c_2 \parallel c_3) + (c_2 \parallel c_4)$$

where any declarations in the composition on the left hand side of the equation are carried out to each of the terms in the right hand side.

7.3 The Brock-Ackerman Anomaly

An algebra of concurrent processes is defined over equivalence classes of the processes.[2] The *canonical members* of each equivalence class provide an abstract representation for all the processes in the class. There are two considerations in defining equivalence relations. On the one hand, the abstract representation of processes must discriminate between systems which when operated on or composed with other systems lead to behaviors we wish to distinguish from each other. On the other hand, the representation must not discriminate between systems that behave identically in all *contexts*. A context is determined by the degree of encapsulation and the "environment" of other processes it interacts with.

In the case of sequential programs, the *history relation* which maps inputs to outputs is sufficient to provide an abstract characterization of a program. In the context of concurrent systems, the history relation is the weakest equivalence relation which may be used to model systems. In other words, it contains the *minimal* information necessary to differentiate between systems. Unfortunately, as [Keller 77] and [Brock and Ackerman 81] have shown, it is not sufficient to discriminate between systems that are observably different. Of the two cases cited, the Brock-Ackerman anomaly

[2] In this section we use the term process to impart a general flavor to the discussion. In particular, systems of actors are "processes."

represents a more serious problem. We discuss it in the context of actor systems.

The Brock-Ackerman anomaly shows that when each of two systems with the same history relations is composed with an identical system, the two resulting combined systems have distinct history relations. Let $\mathcal{H}$ be a function mapping a process to the history relation it defines. We convert the relation into a function by using the standard technique of collecting all the terms representing the possible elements which are related to each given element of the domain. We first define two actor systems S_1 and S_2 such that they have an $\mathcal{H}(S_1) = \mathcal{H}(S_2)$. We then define a system U and show that $\mathcal{H}(S_1 \parallel U) \neq \mathcal{H}(S_2 \parallel U)$ where $\parallel$ represents a parallel composition.

The receptionist in both systems S_1 and S_2 is an actor whose behavior is described by:

```
$D(a)$ $[k]$
    send $[k]$ to $P_i$
    send $[k]$ to $P_i$
```

In other words D accepts a communication and sends two copies of it to an acquaintance a. Its behavior is unserialized. The external actor in both systems S_1 and S_2 is called *extern-acq.* In S_1 the behavior of the acquaintance a is to store the first communication it accepts and to send it and the second communication accepted to *extern-acq.* It can be described as :

```
$P_1$(inputs-so-far, external-acq, first-input) [k]
    if inputs-so-far=0 then become new $P_1$(1, external-acq, k)
    if inputs-so-far=1 then
        become SINK
        send [first-input] to external-acq
        send [k] to external-acq
```

where the behavior of a SINK is simply to "burn" all communications it accepts.

Now a system whose population is $\{d, p\}$, with behaviors $D(p_1)$ and $P_1(0, e, 0)$, respectively, and whose external actor is e, has the history relation which maps:

$$\begin{array}{rcl} \emptyset & \rightarrow & \emptyset \\ \{x_1\} & \rightarrow & \{y_1\, y_1\} \\ \{x_1\, x_2\} & \rightarrow & \{y_1\, y_1\,,\ y_1\, y_2\,,\ y_2\, y_2\} \end{array}$$

where x_i is the communication k_i sent to the target d, and y_i is the communication k_i sent to the target e. Recall the arrival order nondeterminism in actors. Thus $x_1\, x_2$ is the same as $x_2\, x_1$ since the communications may arrive in either order at the target d. Internally, when d accepts $[k_1]$ it will send two k_1 messages to p_1 and similarly for k_2. However, these four communications to p_1 may be interleaved in an arbitrary manner. In general, the history relation can be represented as:

$$x_1 \ldots x_n \rightarrow \{x_i\, x_j \mid 1 \leq i, j \leq n\}$$

Now consider an actor system S_2 with a receptionist d which has an acquaintance p_2. The initial behavior of p_2 is described by $P_2(0, e)$ where:

```
P_2(inputs-so-far, external-acq) [k]
    send [k] to external-acq
    if inputs-so-far=0 then become new P_1(1, external-acq)
    if inputs-so-far=1 then become SINK
```

The difference between the (initial) behavior of p_1 and p_2 is that p_1 waits for two inputs before forwarding them both but p_2 forwards two inputs as they are received. However, as the reader may readily convince themselves, because of arrival order nondeterminism the history relation on the system S_2 is identical to that on system S_1.

Suppose that each of the actor systems S_i are composed with another actor system U where e_1 is the receptionist and has the (unserialized) behavior $E(e_1, e_2)$, where E is as follows:

```
E(external-acq1 , external-acq2) [k]
    send [k] to external-acq2
    send [k] to external-acq1
    send [5 * k] to external-acq1
```

In U both e_1 and e_2 are external. When we compose S_i with U, d is the only receptionist and e_2 the only external actor in the composed system. The external actor a in U is declared to be the receptionist d (see fig 7.3). The history relation on T_1 which is the composition of S_1 and U maps

$$x_1 \rightarrow y_1 \, y_1$$

where y_1 is the message k_1 to e_2. Note that p_1 has accepted both communications *before* forwarding them to e_2. However, the history relation on T_2

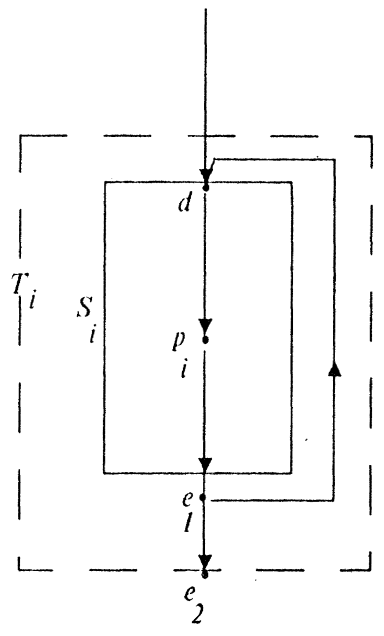

Figure 7.3: *The Brock-Ackerman anomaly. When the systems S_1 and S_2 are composed with a system U which has the population e_1, the history relations of the two composed systems are quite different.*

maps

$$x_1 \rightarrow \{y_1\, y_1\,,\; y_1\, y_1'\}$$

where y_1' is the message $5*k_1$ sent to e_2. This happens because the second k_1 sent to p_2 may arrive *after* the $5*k_1$ message sent by e_1 has been forwarded and accepted by p_1.

The Brock-Ackerman anomaly demonstrates the insufficiency of the history relation in representing the behavior of actor systems (in fact, in any processes which have a nondeterministic merge in them). The problem with the history relation is that it ignores the open, interactive nature of systems which may accept communications from the outside and send communications out at any stage. Having sent a communication, the system is in a different set of possible configurations than it was before it did so, and provided we have a model for the behavior of a system, we can deduce that the number of possible configurations it may be in has been reduced. Thus the two systems, S_1 and S_2 are different to begin with because after having sent a communication to the outside, their response to subsequent communications from the outside is distinct.

7.4 Observation Equivalence

We have seen two equivalence relations on configurations in the previous sections. The first of these was a *direct relabeling* equivalence an the second was the equivalence induced by a *history relation.* Neither of these equivalences is satisfactory. The history relation was shown to be too weak; it collapses too many configurations into the same equivalence class.

The equivalence relation induced by direct relabeling is not satisfactory in an admittedly direct sense. For example, suppose two configurations were identical except that at one mail address m the actors in their respective *states* differed in that only the tags and mail addresses created by them were unequal. (This could happen using our definition of the behavior function if, for example, the order of new expressions was different). Using direct equivalence, these configurations would not be mapped to the same equivalence class. What we would like to do is to consider configurations that have transitions to equivalent configurations equivalent. Fortunately an inductive definition, establishing equivalences to depth n for an arbitrary depth, is not necessary for this purpose: Since there are only a finite number of behavior definitions, their equivalence under relabeling can be directly established as well.

Unfortunately, this weaker relabeling equivalence is not satisfactory either. Consider two configurations which are identical <u>except</u> that one of them has an actor x such that:

1. x is <u>not</u> a receptionist;

2. x is <u>not</u> the target of any task in the configuration; and

3. the mail address of x is <u>not</u> known to any other actor and is <u>not</u> in any of the communications pending in the configuration.

It can be safely asserted that the two configurations, with and without the actor x, are equivalent (see §3.1.1). In implementation terms, the actor x would be a suitable candidate for *garbage collection.* However, these two configurations are clearly not equivalent under relabeling.

We therefore need to define a notion of *observation equivalence* between configurations (following [Milner 80]). The only events "observable" in a encapsulated system are of two kinds:

- Communications sent from the outside to some receptionist; and

- Communications sent by an actor in the population to an external actor.

This suggests three kinds of transitions from each configuration— transitions involving the acceptance of a communication sent from the outside to a receptionist (input), a transition involving the sending of a communication to an external actor (output), and an internal action (corresponding to processing a task in the configuration which is internal to the configuration). The first kind of transition leads to a configuration c' from a given configuration c such that $tasks(c') = tasks(c) \cup \tau$ where τ is the task accepted from the outside. The other two kinds transitions are the ones already defined in chapter 5, except that we ignore the labels on all transitions that are not targeted to an external actor. We can now identify computation in actor systems as a tree with these three kinds of labels on its branches (see Appendix).

How does the composition of trees work in this framework? In CCS, when two trees are combined, the inputs and outputs are matched in a synchronous manner and constitute a silent transition. Rather surprisingly, no change in the technical aspect of this definition is necessary to accommodate composition in actor systems despite the fact that communication is *asynchronous* in actor system. The reason is simply as follows: Only the acceptance of a communication constitutes a transition from a configuration, thus when two configurations are composed all we are doing is *reconciling* the acceptance of a communication by an external actor, with the subsequent behavior of that actor. The latter is given by the actions in the tree corresponding to the configuration where the actor is a receptionist. Because of arrival order nondeterminism, the arrival of the communication is delayed arbitrarily long in the first configuration, thus the composition is, in effect, asynchronous.

A configuration can be extensionally defined using the tree of events specified above. The definition is inductive— two configurations are observation equivalent to degree n if they have have the same observable transitions at the $n^{\underline{th}}$ level of the tree. This notion differentiates between all the configurations one would want to differentiate between. After all, if it is impossible to observe the difference between two configurations despite any interaction one may have with the systems involved, then there is no point discriminating between the two systems.

Brock [83] has proposed a model using *scenarios* which relate the inputs and the outputs of a system using a causal order between them. The model however has several limitations, such as fixed input and output "ports," and it does not support compositionality. The first of these two deficiencies is related to the lack of a labeling scheme such as is afforded by the mail address abstraction in actors.

Philosophically, what we understand to be causality may be nothing more than necessary sequentiality: After all, the pragmatic significance of imputed causal inference in the physical world is simply an expectation of sequentiality in the spatio-temporal order between events considered to be the cause and those considered to be the effect. The inference of all causal relations is an open-ended, undecidable problem since the observation of a cause may be separated from the observation of an effect by an arbitrary number of events. The same *arbitrary delay property* is true of the guarantee of mail delivery. Both of these properties may only be deduced from a proposed model of the internal workings of a system rather than from observations on a system. In contradistinction, the notion of observation equivalence is based on the testability of equivalence to an arbitrary depth.[3]

The problem with the history relation is that it ignored the open, interactive nature of systems. Any system may accept a communication at any time, and given that it has produced a particular communication, its response to a subsequent input is different because of the transitions it has undergone to produce that particular communication. The communication produced is of course simply symptomatic of the change in the system. Internally, the change has already occurred, whether or not we have observed its external manifestation— i.e., whether or not the communication sent has been received. On the one hand, until we observe the effects of the change, there is uncertainty, from the external perspective, as to whether the change has already occurred. On the other hand, after we have observed the effects of a transition, we have at best a model for how the system was at the time the transition occurred rather than a model of its

[3]Admittedly, a curious proposition since we can test only one path of possible evolutions of a system. The usual solution to this difficulty is having an arbitrary number of systems pre-determined to be equivalent, presumably in some stronger physical sense. The idea is to experiment on these systems in different ways to determine their behavior to any desired degree.

"current" status.[4] However, if we have any understanding of the mechanics of a system, given a communication from that system, we can prune the tree of possible transitions that the system may have taken.

[4]Compare the reasoning behind the old *Heisenberg Uncertainty Principle* to the situation here. An interestinng discussion of "quantum physics and the computational metaphor" can be found in [Manthey and Morey 83].

Chapter 8

Conclusions

We have developed a foundational model of concurrency. The model uses very few primitive constructs but can nevertheless accommodate the requirements for a general model of concurrent computation in distributed systems. The flavor of transitions in *actors* is one of a pure calculus for concurrency; it differs from Milner's *Calculus of Concurrent Systems* primarily in two respects: it does not assume *synchronous communication*, and, it explicitly provides for *dynamic creation of agents.*

Actors integrate useful features of *functional programming* and *object-oriented programming.* While other functional systems have some measure of difficulty dealing with *history-sensitive shared objects*, actors do so quite easily. At the same time, actors avoid sequential bottlenecks caused by assignments to a store. The concept of a store, in the context of parallel processing, has been the nemesis of the von-Neuman architectures.

Actors are inherently parallel and exploit maximal concurrency by using the dynamic creation of customers and by *pipelining* the replacement process. The semantics of replacement is fundamentally different from changes to a local store. Replacements may exist concurrently. This kind of pipelining can be a powerful tool in the exploitation of parallel processors. In fact pipelining (specifically, instruction pre-fetching), has been an extremely successful tool in speeding up the computation on many processors currently in use. Unfortunately, the degree to which pipelining can be carried out in the current generation of processors is restricted by the ubiquitous assignments to a store, and the use of *global states* implicit in the program counter. Actors allow pipelining to be carried out to its logical limits as constrained by the structure of the computation and by the hardware resources available.

Perhaps the most attractive feature about actors is that the programmer is liberated from explicitly coding details such as when and where to force parallelism and can concentrate on thinking about the parallel complexity of the algorithm used. If one is to exploit massive parallelism, using parallel processors on the order of tens, perhaps hundreds, of millions of processors, it will not be feasible to require the programmer to explicitly create every process which may be executed concurrently. It is our conjecture that actors will provide the most suitable means for exploiting parallelism made feasible by the advent of distributed systems based on VLSI.

Message-passing is elemental to computation in actors. The time complexity of communication thus becomes the dominant factor in program execution. More time is likely to be spent on communication lags than on the primitive transformation on the data. Architectural considerations such as load balancing, locality of reference, process migration, and so forth, acquire a pivotal role in the efficient implementation of actor languages.

The information provided by a transitional model of actor systems is too detailed to be of "practical" use. The structure of transactions and transaction-based reasoning for the verification of actor programs needs to be studied. The semantics developed here will simply provide the justification for such axiomatic treatment. The open and interactive nature of actors implies that any description of actor behavior will necessarily involve a combinatorial explosion in the exact configurations possible in a system. However, by establishing invariants in the behavior of a actor, we can satisfy our self as to its correctness. The importance of proving program correctness in concurrent systems is underscored by the fact that it is not possible to adequately test such systems in practice. In particular, arrival order nondeterminism implies that any particular sequence of message delivery need <u>never</u> be repeated regardless of the number of tests carried out.

Another critical problem for computer architectures to support actors is controlling computational activity. Since actors may be shared objects, one can not simply assign them a fixed amount of computational resources upon creation. If transactions involving the same actors are concurrently executed, the resources used by each transaction need to be assessed separately. Furthermore, concurrent sub-transactions are spawned dynamically in actor system as many messages may be sent in response to a single message. These sub-transactions must be allocated resources dynamically as well. Since it is impossible to correctly assess the computational resources needed, the allocation has to be constantly monitored. The problem of transactions is in general intractable if the transactions are not properly nested.

We have addressed a number of general problems that plague computation in distributed systems. Among these problems are *deadlock*, *divergence*, *abstraction* and *compositionality.* The problem of deadlock is dealt with by the *universal replacement requirement.* The effects of divergence on the semantics of a computation are contained by the *guarantee of mail delivery.* The problem of abstraction is addressed by the concepts of receptionists and transactions and, at the model-theoretic level, by the notion of *observation equivalence.* And finally we support compositionality using pure *message-passing.*

A simple minimal actor language is shown to be sufficient to accommodate extremely expressive structures, including potentially infinite ones. The denotational semantics of actor behaviors is defined and a transition relation for configuration follows simply from the semantics. Finally, we have dealt with equivalence relations between actors and provided some connections with other models of concurrency.

Appendix A

Asynchronous Communication Trees

Milner [80] has developed an elegant calculus for synchronously communicating agents (called CCS). As an aid to visualizing computation in a system of such agents, Milner has proposed *Communication Trees* (CTs) as a model for CCS. As Milner has observed, CTs are actually more powerful than CCS; in other words, there are large classes of CTs which can not be expressed as programs in CCS. For example, the topology implied by CCS is static whereas there is no such restriction on CTs. We develop *Asynchronous Communication Trees* (Υ's) as a model to aid in visualizing computation in actors and a means by which we can define composition, direct equivalence, observation equivalence, etc., in actor systems. The intriguing feature of Υ's is that they capture the open, interactive nature of computation in actors. It is recommended that the reader carefully study

Milner's work, in particular Chapters 5 and 6 before trying to figure out the material here in any depth.

There are three fundamental differences between actors and CCS:

- Communication is synchronous in CCS while it is asynchronous in actors.
- The topology on CCS agents is static while communications in actors may contain mail addresses.
- There is no dynamic creation of agents in CCS while actors may be created dynamically.

Rather surprisingly, the algebra used to define Υ's is almost identical to that used in CTs; the primary difference is in the concrete interpretations associated with each. We interpret only the *acceptance* of a communication as a transition (what Milner calls "action"): Sending a communication is simply represented by the fact that every branch of the tree has a transition corresponding to the acceptance of the communication by its target. This fact follows from the guarantee of mail delivery.

We represent each configuration as an Υ. A few simplifying assumptions will be made. First, we assume that there are no mail address conflicts between different configurations (since we provide a relabeling operator, this is without loss of generality). Second, we assume that the external mail addresses represent the true mail address of the external actor. When two configurations are composed, this will be a useful simplification. The justification for this assumption is two-fold: firstly, using message-passing the

external actor *forwards* all mail it has received to the actor it is supposed to become; and secondly, the communications it sent to the external actor can arrive in any order in the first place. Thirdly, we assume that there are a countable number of communications, which may be enumerated as $k_0, k_1, \ldots$. Any communication may be sent to any actor, if the communication sent is inappropriate (such as having the wrong number of "parameters"), then we assume there is a default behavior. We can assume that the tags of tasks are part of the enumeration of communications and used to create new mail addresses.[1] However, tagging tasks is not useful in defining observation equivalence; furthermore, it is also possible to specify a different mechanism to create mail addresses using other mail addresses. The technical development remains quite similar.
A typical Υ consists looks like Fig A.1

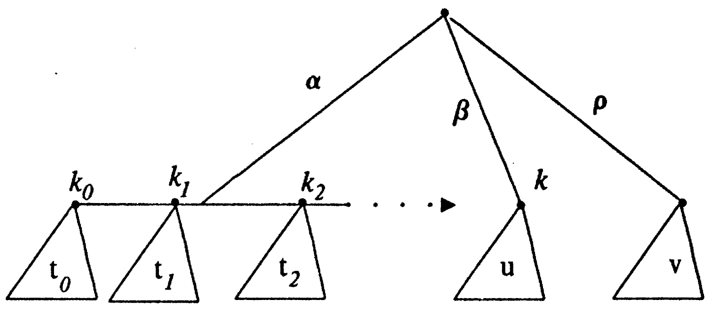

Figure A.1: *A typical Asynchronous Communication Tree*

The three kinds of potential transitions (or in Milner's terminology,

[1]Recall that a union of all finite collections of a countable set is still countable.

actions) have the following intuition:

(i) Corresponding to each current receptionist in the configuration is a *potential transition* labeled by its mail address (these are the positive labels, $\alpha, \ldots$, in the figure) and the communication it may accept. The $i^{\underline{th}}$ tree it dominates represents the behavior of the configuration if the receptionist accepts a communication k_i.

(ii) For all communications accepted by an external actor, there is a transition to the tree corresponding to the behavior of a configuration without the pending communication. These transitions are labeled by the mail address of the external actor (using negative labels $\overline{\alpha}, \ldots,$).

(iii) For all communications accepted by an actor in the population, there is an *internal transition* (labeled by ϱ).[2]

The relation between configurations and Υ's should be intuitively clear from the above description. If each node were marked by the configuration it represented, then (i) would correspond to a transition from a configuration c to a configuration c' such that $states(c) = states(c')$ and $tasks(c') = tasks(c) \cup \tau$ where τ is the task accepted from the outside; (ii) is the acceptance of a communication by an external actor, and (iii) is acceptance of a communication by any actor in the population.

When an actor accepts a communication, it may create other actors or send new communications to specific targets. These will simply show up

[2]Milner represents internal transitions, or what he terms "silent actions," by τ but we used that letter to denote tasks.

in the subsequent behavior of the configuration to which the transition is made. We do not label nodes in our trees (cf. Milner's CTs) because the label would be simply another way of noting the sub-tree dominated by the given node. Formally we define an ACT as follows:

Definition A.1 Asynchronous Communication Trees. *Assume the function extern(k) represents the external actors communicated in k. An Υ with receptionists R and external actors E is a finite set*[3] *of pairs of the form*

(i) $< \alpha, f >$ $\alpha \in R$ *where f is a family of Υ's with receptionists R and external actors $E \cup extern(k_i)$ indexed by possible communications k_i accepted; or,*

(ii) $< \overline{\beta}, < k, t >>$ $\overline{\beta} \in E$ *where k is a communication targeted to the mail address β and t is an Υ with receptionists R and external actors $E \cup extern(k)$; or,*

(iii) $< \varrho, t >$ *where t is an Υ with receptionists R and external actors E.*

Remark. The receptionists denote all the current receptionists as well as the (potential) future receptionists; the external actors denote only the currently known external actors. The asymmetry arises because any future receptionists are created locally thus their potential mail addresses are <u>internally</u> determined (even if they are functions of the incoming tags),

[3]We do not need a multiset because all mail addresses are unique. However, Υ's in their full generality may lack rules guaranteeing uniqueness of mail addresses. Technically this does not create any difficulty; it simply changes the nature of nondeterminism.

however the mail addresses of external actors which become known as a result of an incoming communication are in principle unknowable.

We will now define an algebra of Υ's and then define three operations namely, *composition*, *restriction*, and *relabeling* on the Υ's. The actual algebra is almost identical to CTs except, not surprisingly, in the notion of receptionists and external actors. CTs used sorts which were a fixed set for each CT. The concrete interpretations placed on the terms are, of course, quite different. The definitions below are adapted from [Milner 80].

Let $\Upsilon_{R \times E}$ denote the Υ's with receptionists R and external actors E and $k_0, k_1, \ldots,$ denote the possible communications. We have an algebra of Υ's as follows:

NIL(nullary operation)

NIL is the Υ $\bullet$

$+$ (binary operation)

$$+ \;\in\; \Upsilon_{R_1 \times E_1} \times \Upsilon_{R_2 \times E_2} \;\rightarrow\; \Upsilon_{R \times E}$$

where $R = (R_1 \cup R_2)$ and $E = (E_1 \cup E_2)$

α (a ω-ary operation)

α takes a set of members of $\Upsilon_{R \times E}$ indexed by $k_0, k_1, \ldots,$ and produces a member of $\Upsilon_{(R \cup \{\alpha\}) \times (E \cup extern(k_i))}$ for the kth member. This operation adds a receptionist with mail address α, see Fig A.3. Let $K =$

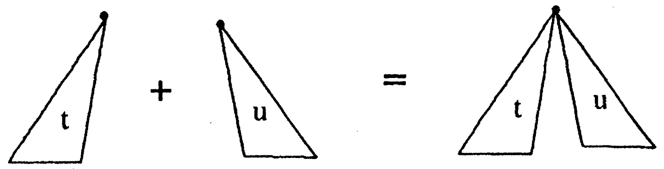

Figure A.2: *Possible nondeterministic transitions.*

$\{k_0, k_1, \ldots\}$, then

$$\alpha \in (K \to \Upsilon_{R \times E}) \to \Upsilon_{(R \cup \{\alpha\}) \times E}$$

$\overline{\alpha}$ (a family of unary operations)

$$\forall k\ \overline{\alpha}(k) \in \Upsilon_{R \times E} \to \Upsilon_{R \times (E \cup extern(k))}$$

$\overline{\alpha}(k)$ represents the fact that the communication k has been accepted by an external actor with the mail address α. See Fig A.4

ϱ (a unary operation)

$$\varrho \in \Upsilon_{R \times E} \to \Upsilon_{R \times E}$$

The interpretation of the + operation is nondeterminism in the model—$t_1 + t_2$ simply means that we may be in the tree t_1 or in the tree t_2. The rest of the operations are straight-forward and correspond to their description in the introduction. The trees differ from the transitions model of Chapter 5 in

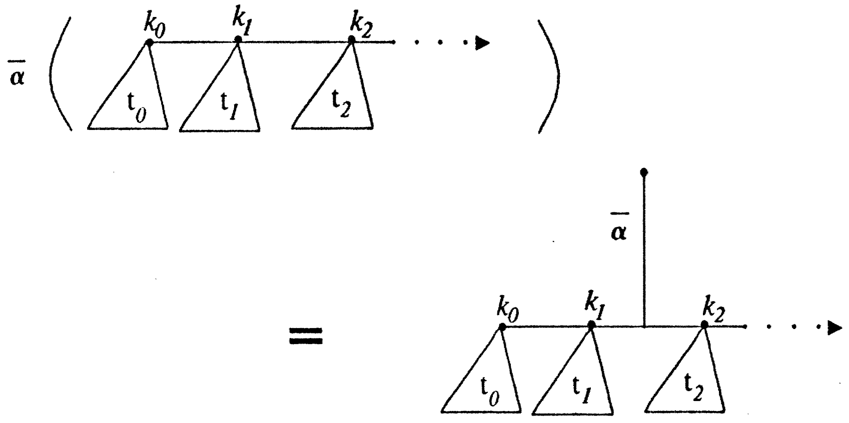

Figure A.3: *A new receptionist definition.*

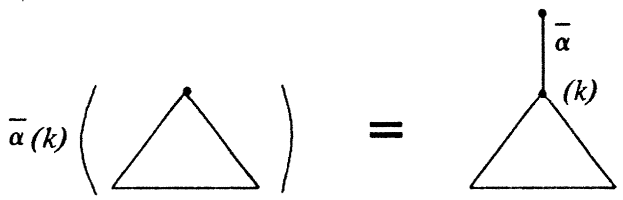

Figure A.4: *Acceptance of a communication by an external actor.*

that they represent computation in an open system: It is possible to send a communication from the outside to a receptionist (the potential transitions are included to account for this fact). However configurations contain all the information necessary to map them into the trees and, furthermore, we can commute the diagram of the composition maps in configurations and Υ's.

The mechanics of how Υ's are built will become clearer with the composition operation. Composition will allow us to combine acceptance of communications sent to the receptionists (positive label bindings) in one tree with acceptance by the corresponding external actors (negative label bindings) in a different tree to create an internal action of the composite. We now provide three operations on Υ's, namely, (concurrent) composition, restriction and relabeling.

Composition

Composition, $\|$, is a binary operation on Υ's such that:

$$\| \in \Upsilon_{R_1 \times E_1} \times \Upsilon_{R_2 \times E_2} \rightarrow \Upsilon_{R \times E}$$

where $R = (R_1 \cup R_2)$, $E = (E_1 - R_2) \cup (E_2 - R_1)$ and $(R_1 \cap R_2) = \emptyset$. Let $t \in \Upsilon_{R_1 \times E_1}$ and $u \in \Upsilon_{R_2 \times E_2}$, then $t \| u$ has the following branches:

(i) For each branch of t representing a communication from the outside to a receptionist (i.e., for the branches with the positive labels), there is a branch which represents the input followed by the composition with u of each of the trees it dominates. This branch reflects the fact that communications may be received by receptionists in t before any other actions take place. *Mutatis mutandis* for the receptionist of u.

(ii) For each branch of t representing an internal transition, a branch corresponding to the internal transition followed by the composition with u of the tree it dominates. This simply says that the internal action could happen before any of the effects of composition happen. *Mutatis mutandis* for branches of u.

(iii) For each branch of t representing a communication to an external actor β there are two possibilities. If $\beta \notin R_2$ then there is simply a equivalent branch followed by the composition with u of the tree it dominates. Otherwise, for each branch of u representing a communication from the outside to the receptionist β, there is an internal action followed by the composition of the tree in u which follows accepting the given communication and the tree the "output" branch dominates. The acceptance has been internalized because of the composition. *Mutatis mutandis* for "output" branches of u.

The composition operator preserves arrival order nondeterminism since it simply represents the interleaving of all the possibilities.

Restriction

The *restriction* operation, \, removes a receptionist from a system. The result is that the mail address removed is no longer available for composition with other trees. However, if the corresponding actor was involved in accepting any communications from actors within the configuration, then these transitions are unaffected. One obviously can not remove internal actions since they are not "guarded" by the mail address. Formally,

$$\backslash\alpha \in \Upsilon_{R \times E} \rightarrow \Upsilon_{(R - \{\alpha\}) \times E}$$

Relabeling

Given a map from mail addresses to mail addresses, this operator changes both the positive and negative bindings associated with each. It is unary operator. Note that in an actor system, $R \cap E = \emptyset$, therefore positive and negative versions of the same label can not co-exist in the same Υ.

We skip the straight-forward recursive definitions of restriction and relabeling.

The algebra now behaves like the algebra of CTs; in particular, the same definitions of strong equivalence and observation equivalence can be used. Observation equivalence on Υ's provides an intuitively abstract description of actor systems and retains the right amount of information. We refer to [Milner 80] for details.

An interesting, and not too difficult exercise, is to draw sufficient fragments of the Υ's for the two systems S_1 and S_2 used in discussion of the Brock-Ackerman anomaly (§7.3). These Υ's are indeed not observation equivalent.

One remark may be pertinent, if peripheral, here. Milner has shown that observation equivalence is a congruence relation for all operations except the "+" operator. The counter-example which shows that observation equivalence is not a congruence relation uses the absorption property of the NIL tree under the + operation. The counter-example would not work if NIL had internal transitions to NIL. In any case, a congruence relation can be defined in terms of observation equivalence.

References

[Ackerman 84] Ackerman, W. B. Efficient Implementation of Applicative Languages. LCS Tech Report 323, MIT, March, 1984.

[Agerwala and Arvind 82] Agerwala, T. and Arvind. Data Flow Systems. *Computer 15*, 2 (Feb 1982).

[Agha 84] Agha, G. Semantic Considerations in the Actor Paradigm of Concurrent Computation. Proceedings of the NSF/SERC Seminar on Concurrency, Springer-Verlag, 1984. Forthcoming

[Agha 85] Agha, G. Actor Information Systems. M.I.T. A.I. Lab

[Atkinson and Hewitt 79] Atkinson, R. and Hewitt, C. Specification and Proof Techniques for Serializers. IEEE Transactions on Software Engineering SE-5 No. 1, IEEE, January, 1979.

[Backus 78] Backus, J. Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM 21*, 8 (August 1978), 613-641.

[Brinch Hansen 77] Brinch Hansen, P. *The Architecture of Concurrent Programs.* Prentice-Hall, Englewood Cliffs, N.J., 1977.

[Brock 83] Brock, J. D. A Formal Model of Non-determinate Dataflow Computation. LCS Tech Report 309, MIT, Aug, 1983.

[Brock and Ackerman 81] Brock J.D. and Ackerman, W.B. Scenarios: A Model of Non-Determinate Computation. In *107: Formalization of Programming Concepts*, Springer-Verlag, 1981, pp. 252-259.

[Brookes 83] Brookes, S.D. A Model For Communicating Sequential Processes. Tech. Rep. CMU-CS-83-149, Carnegie-Mellon, 1983.

[Clinger 81] Clinger, W. D. Foundations of Actor Semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May, 1981.

[Cook 81] Cook, S.A. Towards a Complexity Theory of Synchronous Parallel Systems. *L'Enseignement Mathematique, Reveu Internationale, Geneva* (Jan-June 1981).

[Costa and Stirling 84] Costa, G. and Stirling, C. A Fair Calculus of Communicating Systems. Foundations of Computer Theory, LNCS, Springer-Verlag, 1984.

[Dahl, Myhrhaug, and Nygaard 70] Dahl O. J., Myhrhaug B., and Nygaard K. Simula Common Base Language. Tech. Rep. S-22, Norwegian Computing Center, October, 1970.

[Date 83] Date, C.J. *An Introduction to Database Systems.* Addison-Wesley, 1983.

[de Bakker 80] de Bakker, J.W. *Mathematical Theory of Program Correctness.* Prentice-Hall International, 1980.

[de Bakker and Zucker 82] de Bakker, J.W. and Zucker, J.I. Processes and the Denotational Semantics of Concurrency. *Information and Control* , 54 (1982), 70-120.

[Dijkstra 77] Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, 1977.

[Emden and Filho 82] van Embden, M.H., and de Lucena Filho, G.J. Predicate Logic as a Language for Parallel Programming. In *Logic Programming,* Academic Press, 1982.

[Feynman et al 65] Feynman, R., Leighton, R., and Sands, M. *The Feynman Lectures on Physics.* Addison-Wesley, 1965.

[Golson and Rounds 83] Golson,W. and Rounds,W. Connections Between Two Theories of Concurrency: Metric Spaces and Synchronization Trees. *Information and Control*, 57 (1983), 102-124.

[Gray 80] Gray, J. Experience with the System R Lock Manager. IBM San Jose Research Laboratory, 1980.

[Greif 75] Greif, I. Semantics of Communicating Parallel Processes. Technical Report 154, MIT, Project MAC, 1975.

[Gurd, et al 85] Gurd, J.R., Kirkham, C.C., and Watson, I. The Manchester Prototype Dataflow Computer. *Communications of the ACM 28,* 1 (January 1985), 34-52.

[Harel 79] Harel, D. *Lecture Notes in Computer Science. Vol. 68: First-Order Dynamic Logic.* Springer-Verlag, 1979.

[Henderson 80] Henderson, P. *Functional Programming: Applications and Implementation.* Prentice-Hall International, 1980.

[Hewitt 77] Hewitt, C.E. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence 8-3* (June 1977), 323-364.

[Hewitt 80] Hewitt C. E. The Apiary Network Architecture for Knowledgeable Systems. Conference Record of the 1980 Lisp Conference, Stanford University, Stanford, California, August, 1980, pp. 107-118.

[Hewitt 83] Hewitt, C. Some Fundamental Limitations of Logic Programming. A.I. Memo 748, MIT Artificial Intelligence Laboratory, November, 1983.

[Hewitt and Atkinson 77] Hewitt, C. and Atkinson, R. Synchronization in Actor Systems. Proceedings of Conference on Principles of Programming Languages, January, 1977, pp. 267-280.

[Hewitt and Baker 77] Hewitt, C. and Baker, H. Laws for Communicating Parallel Processes. 1977 IFIP Congress Proceedings, IFIP, August, 1977, pp. 987-992.

[Hewitt and de Jong 83] Hewitt, C., de Jong, P. Analyzing the Roles of Descriptions and Actions in Open Systems. Proceedings of the National Conference on Artificial Intelligence, AAAI, August, 1983.

[Hewitt, et al 84] Hewitt, C., Reinhardt, T., Agha, G. and Attardi, G. Linguistic Support of Receptionists for Shared Resources. Proceedings of the NSF/SERC Seminar on Concurrency, Springer-Verlag, 1984. Forthcoming

[Hoare 78] Hoare, C. A. R. Communicating Sequential Processes. *CACM 21*, 8 (August 1978), 666-677.

[Holland 75] Holland, J.H. *Adaptation in Natural and Artificial Systems.* U. of Michigan Press, 1975.

[Hwang and Briggs 84] Hwang, K. and Briggs, F. *Computer Architecture and Parallel Processing.* McGraw Hill, 1984.

[Kahn and MacQueen 78] Kahn, K. and MacQueen, D. Coroutines and Networks of Parallel Processes. Information Processing 77: Proceedings of the IFIP Congress, IFIP, Academic Press, 1978, pp. 993-998.

[Keller 77] Keller, R.M. Denotational Models for Parallel Programs with Indeterminate Operators. Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts, IFIP, August, 1977.

[King and Collmeyer 73] King, P. and Collmeyer, A. Database Sharing: An Efficient Mechanism for Supporting Concurrent Processes. Proceedings of NCC, 1973.

[Liskov, Snyder, Atkinson, and Schaffert 77] Liskov B., Snyder A., Atkinson R., and Schaffert C. Abstraction Mechanism in CLU. *Communications of the ACM 20*, 8 (August 1977).

[Lynch and Fischer 81] Lynch, N. and Fischer, J. On Describing Behavior and Implementation of Distributed Systems. *Theoret. Comp. Science 13*, 1 (1981).

[Manthey and Moret 83] Manthey, M. and Moret, B. The Computational Metaphor and Quantum Physics. *CACM* (February 1983).

[McCarthy 59] McCarthy, John. Recursive Functions of Symbolic Expressions and their Computation by Machine. Memo 8, MIT, March, 1959.

[Mead and Conway 80] Mead, C. and Conway, L. *Introduction to VLSI Systems.* Addison-Wesley, Reading, MA, 1980.

[Meijer and Peeters 82] Meijer, A. and Peeters, P. *Computer Network Architectures.* Computers Science Press, 1982.

[Milner 80] Milner, R. *Lecture Notes in Computer Science. Vol. 92: A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[Peterson 77] Peterson, J.L. Petri Nets. *Comput. Survey* (Sept. 1977).

[Pnueli 83] Pnueli, A. On the Extremely Fair Treatment of Probabilistic Algorithms. Proceedings of the Fifteenth Annual ACM Symposium on the Theory of Computing, 1983.

[Pratt 82] Pratt, V. R. On the Composition of Processes. Proceedings of the Ninth Annual ACM Conf. on Principles of Programming Languages, 1982.

[Scott 72] Scott, D. S. Lattice Theoretic Models for Various Type-free Calculi. Proceedings 4th International Congress in Logic, Methodology and the Philosophy of Science, Bucharest, Hungary, 1972.

[Scott 82] Scott, D. S. Domains for Denotational Semantics. ICALP-82, Aarhus, Denmark, July, 1982.

[Seitz 85] Seitz, C. The Cosmic Cube. *Communications of the ACM 28*, 1 (January 1985), 22-33.

[Smyth 78] Smyth, M.B. Petri Nets. *J. of Comput. Survey Science* (Feb. 1978).

[Stoy 77] Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, MA, 1977.

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

[von Neumann 58] von Neumann, J. *The Computer and the Brain.* Yale U. Press, New Haven, Conn., 1958.

[Weng 75] Weng, K.-S. Stream-Oriented Computation in Data Flow Schemas. TM 68, MIT Laboratory For Computer Science, October, 1975.

[Wirth 72] Wirth, N. The Programming Language Pascal. Eidgenossiche Technische Hochschule Zurich, November, 1972.