Massachusetts Institute of Technology
Artificial Intelligence Laboratory

A.I. Memo No. 672 April, 1982

A Primer for the Act-1 Language

Dan Theriault

Abstract

This paper describes the current design for the *Act-1* computer programming language, and describes the Actor computational model, which the language was designed to support. It provides a perspective from which to view the language, with respect to existing computer language systems and to the computer system and environment under development for support of the language. The language is informally introduced in a tutorial fashion and demonstrated through examples. A programming strategy for the language is described, further illustrating its use.

Key words and phrases: actor; concurrency; message passing; parallelism; programming language; programming language system.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505 and in part by the Office of Naval Research under Office of Naval Research contract N00014-75-C-0522.

Document History

A generation of this document was submitted to MIT's Department of Electrical Engineering and Computer Science in June of 1981 in partial fulfillment of requirements for a Bachelor of Science degree. It was dedicated to my wife, Candace. The paper also saw light as MIT AI Working Paper 221.

Acknowledgements

I would like to take this opportunity to thank my thesis supervisor, Carl Hewitt. He spent much time and effort answering my questions, reading drafts of this document, and making detailed and constructive comments. I would also like to thank Giuseppe Attardi for stimulating and enlightening discussions about all aspects of the Act-1 language system. Thanks to the Message Passing Semantics group for their excellent work in conceiving and designing the Act-1 language system. I am indebted to Robert Baldwin for poring over a few drafts and making comments on its consistency, style, and content. I would like to thank Candace Theriault for moral support, for pointing out areas which were not clearly explained, for constructive comments about composition, and for prodding me to work on this document. Finally, I would like to express my gratitude to the members of the Computer System Structure and Computer Systems and Communication groups, in particular, Robert Baldwin, David Reed, Karen Sollins, Muriel Webber, and Debby Fagin, for their moral support and for providing such a stimulating and comfortable environment to work in.

Table of Contents

Chapter One

A Guide to this Document

1.1 Introduction

This document describes the current design for the computer programming language, *Act-1*, which is based on the *Actor* model of computation [Hewitt and Baker 78, Clinger 81]. An overview of the model introduces the major concepts in the language. Act-1 includes both an implementation language and a corresponding descriptive language for actors. The descriptive portions of the language consist of a general knowledge representation system called *Omega* [Hewitt, Attardi and Simi 80, Attardi and Simi 81]. The *Ether* problem-solving system [Kornfeld and Hewitt 81] is used by Act-1 language translators, to reason from information stored by Omega. The Act-1 language is inherently concurrent, and can exploit parallelism in systems of cooperating computers. A highly parallel computer architecture, called the *Apiary* [Hewitt 80] is being studied as a substrate for the language. Such a substrate will realize the potential of the Act-1 language, and help in the quest for discovering implementation techniques suitable for programming in a highly parallel environment.

Act-1, Omega, Ether, and the Apiary have been under parallel development by MIT's Message Passing Semantics Group. Preliminary implementations of Omega, Ether and Act-1 have been written to aid in their development. This document informally presents the design of Act-1, its philosophy and use, in a tutorial fashion. Its emphasis is on presentation rather than on completeness and formality. A more formal definition of the language and discussion of subtle issues will be presented in a document by Hewitt and Attardi, to appear.

1.2 Organization

The style and structure of this document reflect a progressive refinement method of exposition.

Chapter 1 puts this document into historical perspective, and describes the structure of the document. It has little technical content.

Chapter 2 discusses the actor computational model, and in so doing, describes the major ideas in (foundations of) Act-1.

Chapter 3 describes the Act-1 language system.

Chapters 4 and 5 present an overview of the language, Chapter 4 presenting the descriptive constructs of Act-1; Chapter 5, an overview of the imperative constructs.

Chapters 6 through 8 describe the details of the language. Chapter 6 enumerates the portions dealing with transmission of communications between actors. Chapter 7 describes the fundamental concept of behavior, and its expression in the language. Chapter 8 describes the language constructs not yet discussed, and summarizes the rest.

Chapters 9 and 10 describe the implementation of actors in the language, and in so doing review the language constructs.

Chapter 11 describes a methodology for programming in the language, putting the use of its specification and implementation techniques into perspective.

Chapter Two

The Actor Model of Computation

2.1 Other Established Computational Models

We will set the stage by briefly characterizing more common computational models and introducing the concepts of concurrency and parallelism. In that context, we will proceed with a description of the Actor computational model.

2.1.1 Sequential Computer (or Container-Oriented) Models

Early computing languages had an implicit computational model in which each computation step consisted of a partial modification to a global machine state. Data "types" and "structures" were merely templates with which to interpret parts of the global state. In a machine-oriented model, computation proceeds by sequentially modifying portions of the machine's state. Although languages based on this model typically have a well-developed notion of procedural abstraction, further discussion of them will be limited, in favor of discussion of the object-oriented model, which contains a superset of its abstraction mechanisms. Because the computational paradigm is the modification of a global state, limited potential exists for concurrency, because of the large amounts of synchronization necessary to protect the state.

2.1.2 The Object-Oriented Model

The object-oriented model is based on the notion of changing the state of capsules of information by applying standard operations to them. There exist

different types or classes of capsules. Each has its own set of primitive or standard operations. The major conceptual advance of the model is that of encapsulating the representation of an abstract type in the primitive operations of the type. The underlying computational paradigm is still that of applying operations to cause a state transition.

The traditional sequential Object-Oriented computational model, as exemplified by languages such as Simula [Nygaard and Dahl 66], Clu [Liskov, et al 79], and Alphard [Wulf, London and Shaw 76] supports two kinds of abstractions, procedural and data.

Procedural abstractions represent the performance of some computation and are built using control structures, such as statements for decision or iteration, and primitive procedures or other procedural abstractions.

A data abstraction represents a type of object and is characterized by a representation for objects of that type and a set of primitive operations which may be applied to objects of that type. The representation is in terms of primitive data types or of other abstract data types. The representation of the abstract data type is encapsulated in such a fashion that only the type's primitive operations can contain knowledge of its representation. The primitive operations are the only operations which may be directly applied to objects of that type. They can include operations for creating objects, for extracting information from objects, for comparing objects, for copying objects, or for altering the information contents of objects.

This encapsulation of representation is one of the most important features of the object-oriented computational model. It allows the treatment of abstract data types in two fashions. Users of the abstraction see it as if it were a primitive data type with a given set of operations which can be applied to objects of that type.

Users neither have nor need access to the representation of the data type. Implementors of the abstraction are the programmers who create or maintain the representation and primitive operations of the data type. Implementors are free to change the representation of the data type and the contents or structure of its primitive operations. Changes such as these and addition of new primitive operations have no effect on existing code which uses the data type, as long as the expressions for calling the primitive operations and the external behavior of the primitive operations have not changed.

2.2 Concurrency in Computational Models

Advancement in computer hardware technology in recent years has made the concept of parallel processing increasingly practical. As a result, concurrency has become increasingly important in the design of computational models and languages. Here, we use the meanings of *parallelism* and *concurrency* as defined in [Hewitt and Baker 78]: Two activities are said to be concurrent if no necessary ordering exists on the times at which they occur. That is, either may happen first, or they may happen at the same time. Parallelism is the physical realization of concurrency, in which activities can overlap in time. The distinction is that parallelism need not occur for activities to be concurrent.

Limited forms of concurrency have existed for a relatively long time in computer systems. Asynchronous physical devices, such as input and output devices and storage devices, are perhaps the oldest and simplest forms. Time-multiplexing of physical computing resources ("time-sharing" computing systems) gave the illusion of sole possession of a computer with shared files.

Time-multiplexed computation and shared storage allowed concurrent users

(processes) to communicate. Diverse mechanisms were devised for interprocess communication, process synchronization, and synchronized access to shared resources. For example, these mechanisms include semaphores and monitors.

Some benefits of concurrency in programming languages became apparent. Some computational problems are inherently concurrent, such as searching, VLSI design-rule checking, and weather prediction. Machines were still sequential, so languages had no need to deal with parallelism. The fork/join and co-routine mechanisms were developed.

Developments in hardware technology allowed a computer to contain a small number of central processing units. Multi-processing developed (tightly coupled machines). People began thinking about the language issues involving the expression of such concurrency in languages, or about how a compiler or run-time system could generate the parallelism from sequential programs (as for the Illiac IV).

A lingering problem with machine-oriented and object-oriented languages is that they have not abstracted away the notion of flow of control, which is an inherent barrier to concurrency. As a result, they have done no better than to simulate parallelism using such mechanisms as the coroutine.

Further and more recent developments have made the possibility of large numbers of communicating processors a reality. Networks of computers have been designed and implemented. Communicating Sequential Processes [Hoare 78] models several physical computers cooperating, in a synchronized fashion, to perform a single task. Its computational model is one temporally interleaving the instruction streams of several sequential computers.

The concept of a computer which contains large numbers of small, cooperating processors is not remote. The idea of having many active computing entities processing in parallel is not an impractical one now. In such a scheme, each computational entity has its own processing power, so there is no notion of flow of control. Instead, there is the notion of cooperation; of communication between entities which are under their own control.

2.3 The Actor Computational Model

The Message Passing Semantics group at MIT has been developing a computational model based on the notion of computational entities called *actors.* An actor is a computational entity which communicates with other actors using message-passing. An actor consists of a current behavior, which determines what actions it will take in response to communications it receives. Associated with each actor's name is a mailing address, with which other actors can refer to it or send it communications.[1] The concept of *behavior* is very important in the actor model. It represents a unification of both procedural abstraction and data abstraction into a single abstraction mechanism. Behavior is described in more depth in Chapter 7 (which begins on page 53). For the time being, the following descriptions should prove sufficient to generate an intuitive feel for its meaning.

Each actor's current behavior describes which communications it will accept. For each such communication, it describes what the actor will do to process the communication, which can include creating new actors, sending communications to

---

[1]For example, in a system of computers, each computer may be viewed as an actor (with a very complex behavior!). Each data entity existing in the virtual memory space of each computer can be an actor. Each program or procedure entity existing in the memory space of each computer can be an actor. Even the hardware components which make up each computer can be thought of as actors, which send electrical communications down wires.

other actors (or possibly to itself), and designating the behavior which will replace its current behavior.

A computation is begun by sending a *communication* to an actor, requesting that it perform the computation. An *event* is said to occur when an actor accepts a communication. The computation continues as this actor creates new actors and sends messages to itself or to other actors. Note that because computation is event driven, actors which are not processing communications need no processing power.

A *communication* is, itself, an actor, which is a standard package for information to be sent to an actor. The transmission of communications is treated as a primitive operation in this computational model. An actor wishing to send a communication to another actor need only invoke the *SendTo* primitive to have it sent. The *SendTo* operation will send the communication to its destination. The mailing system is assumed to be reliable, and to deliver the communication to its proper destination, which is called the *target* actor. Transmission of communications is buffered. That is, the mailing system enqueues the communication on a First-In, First-Out buffer from which the target actor removes communications. When an actor's incoming-communication buffer is empty and it is not processing a message, the actor just lies dormant. Otherwise, it dequeues a communication from the front of the incoming-communication queue and processes it.

Note that this method for transmission of communications precludes the bottleneck of rendezvous communication (two-way conversation). Actor communication is unsynchronized and buffered. Unlike synchronous communication in systems like CSP, an actor sending a communication to a target actor need not wait until the target actor is ready to accept the communication. The sending actor simply gives the communication to the mailing system, which queues

it for the target actor, whereupon the sender can continue processing. The activity of processing a communication is pipelined, where the pipelining stages are communication-bound. This is best described by example.

Consider two actors, **Permutations** and **Factorial**. **Permutations** is a naive implementation of an algorithm for finding the permutations of **n** items taken **k** at a time, where **n** and **k** are whole numbers and $n \geq k$. It divides **n** factorial by **n - k** factorial. **Factorial** computes the factorial of a whole number. **Permutations** accepts a communication requesting it compute the permutations of **5** (**n** is bound to **5**) choose **3** (**k** is bound to **3**) and send the result to some *customer*, **c**, which will accept the result and do something with it, such as send it to a printer.

Permutations sends a communication to **Factorial** requesting it compute the factorial of **n**. **Permutations** dynamically creates a customer, which it includes in the communication, that accepts a whole number, divides it by the factorial of **k** (**k = 3**), then sends the result to the customer **c** of the original request to **Permutations**. As soon as **Permutations** has sent that communication, it can accept and process another communication. In the mean-time, **Factorial** can accept the communication sent it by **Permutations**, compute the factorial requested (**5!**), then send the result (**120**) to the customer in the request. This customer will proceed to divide **120** by the factorial of **3** then send the result to **c**, in a manner similar to that just described.

It is sometimes more convenient for programmers to think about, and for us to explain, the interactions of actors as two-way conversations, rather than in terms of a pipelined series of one-way communications. Informal discussion in this document will be in terms of two-way communication.

2.4 Analogies between Actor and Object Models

The actor model unifies the concepts of procedural and data abstractions. Let us explore the analogies between actors and common abstractions in other models.

2.4.1 Procedural Abstractions

A procedural abstraction in conventional languages can be implemented as an actor of the following form: an actor posing as a procedure is "invoked" by sending it a communication requesting that it act on a specified parameter and respond to a specified "customer." Both the parameter and the customer are actors. The actor accepts the communication, does some processing, then sends a communication to the customer in response to the request. The response contains information either acknowledging normal completion of processing and specifying an actor which is the return value, or complaining that it could not complete the processing and specifying an actor containing information relevant to the abortion of processing.

A simple example is a procedure which we can call SumFromZero which returns the sum of all Whole Numbers from zero to some specified whole number, upperBound. Such a procedure can be implemented by an actor named `SumFromZero`. This "procedure" is "invoked" by sending a communication to `SumFromZero` requesting it calculate the sum from zero to `upperBound`, and send the result to a specified customer, `c`. When `SumFromZero` receives the communication, it calculates the sum of the whole numbers from zero to `upperBound`, and sends a communication in reply to the customer, `c`, of the request.

2.4.2 Data Abstractions

On Behavior Modification

Notice that an actor behaving like a simple procedural abstraction does not change its behavior as a result of accepting a request ("being invoked"). That is, the actor `SumFromZero` continues to accept a whole number and reply the sum from zero to that number, no matter what requests it accepts. In contrast, consider an actor which is serving as a data structure, for example, a checking account. Such an actor would accept at least three kinds of communications, a request for its current balance, a request that it make note of a deposit, and a request that it make note of a withdrawal. In response to the first, it would reply with the current balance. In response to each of the other two, it would take on a new behavior to reflect the deposit or withdrawal.

Data Abstractions

A data abstraction consists of a representation for an abstract data type and a set of primitive operations which provide the sole access to the representation. The idea of a data object can be generalized to an actor whose behavior is such that it responds in a corresponding way to communications corresponding to the primitive operations of the object's data type.

For example, consider a Stack data object, for Last-In First-Out storage and retrieval of objects. Primitive operations for stacks are creation of an empty stack, a "push" operation for storing a new value into the stack, a "pop" operation for retrieving and removing the latest value stored in the stack, and a "top" operation for retrieving but not removing the latest value stored in the stack.

Such a Stack data object can be implemented by an actor, `Stack0`[2], whose

[2]We call the actor `Stack0` here to distinguish between one particular Stack actor, and the generalized concept of behavior known as `Stack`.

behavior is as follows. Initially, `Stack0` responds with a complaint to `top` or `pop` requests. When `Stack0` receives a request to `push` an actor, `x`, on top of itself, it replaces its behavior with a new behavior which accepts and processes the `top`, `pop`, and `push` requests in the following manner. If it accepts a `top` request, it replies with the actor, `x`. If it accepts a `pop` request, it replies with the actor, `x`, and replaces its behavior with its original behavior (i.e. with the effect of undoing of the push which gave it its current behavior).

2.5 Some Advantages of the Actor Model

2.5.1 The Actor Model Inherently Supports Concurrency

It is evident that the actor model inherently supports concurrent activity. Recall the description of `SumFromZero`'s caller. The pipelining effect of using one-way communication and customers is an important and frequently-occurring source of concurrent activity.

Even more potential for parallelism exists for actors whose behaviors are constant, since these can be replicated any convenient number of times. Actors such as `SumFromZero`, `Factorial`, and `Permutations`, whose behaviors allow for no behavioral change can be arbitrarily replicated, and copies can be distributed to different computers in a system. Each copy of such an actor can accept and process communications, increasing potential for parallelism.

With the ease of expression of concurrent computation in this model, more algorithms will be written exploiting the concurrency. For example, an actor to compute the factorial of a given number can make use of another actor, RangeProduct, which finds the product of a range of whole numbers.

RangeProduct can recursively decompose a problem of calculating the product of a large range of numbers into a product of two smaller range products. These can be computed concurrently, in much the same fashion. With a sufficient number of processors, such a parallel factorial can execute in the log of the time needed for the execution of a sequential implementation.

2.5.2 Absolute Containment of Behavior

Another important advantage of this model is that of absolute containment. That is, each actor is the guardian of its own behavior. Modification of that state can only be done by the actor itself, at the request of other actors. As part of the processing of a communication, an actor may ask for authentication, for resources, etc., or may refuse to process the communication altogether. Conceptually, under no circumstances can one actor directly read or modify the state of another.

Chapter Three

An Actor-Based Language System

3.1 No Language is an Island

No programming language can stand alone, without an environment of utilities to support it. It needs at least:

- ~ a computer architecture to present it with an interface to the hardware on which it is implemented.
- ~ editors with which a programmer can introduce new abstractions or modify existing ones.
- ~ a filing system in which to store persistent abstractions and instances of those abstractions.
- ~ compilers or interpreters to derive code runnable by the computer architecture from code written in the language.
- ~ support such as standard abstractions or a run-time environment.

Richer language systems provide more sophisticated utilities, such as abstraction libraries, optimization tools, debugging tools, consistency checkers, and even verifiers [Horsley and Lynch 79, Brooks 75].

It is useful to think of an environment in terms of services, rather than in terms of tools, to help liberate our minds of the tendency to dwell on tools we've seen before. There are many useful services which can be performed. Enumerating them, then regrouping and abstracting from them can lead to a more powerful, compatible, and synergistic set of utilities which can be merged into a language

system. Such a design method tends to reduce complexity and duplication of effort in the language system and increase the conceptual unity of the system.

We do not mean to imply that the language cannot exist without an elaborate support system. A computational model can be implemented by more than one kind of language system. The machine-oriented model has been implemented by languages with widely varying syntax and expressiveness. When a language is not accompanied by a support system, programmers tend to construct utilities and scaffolding to support their programming activities, as they feel a need for such tools. Needless to say, this leads to incompatible tools, duplication of effort, and complexity.

3.2 Design of One Actor Language System

One implementation of an actor-based language system has been developed by MIT's Message Passing Semantics group. It provides a number of services, including the following. Memory management services allocate, relocate, and garbage-collect actors. The activities of actors are supported by providing processing power and memory for them, and by transmitting and buffering communications sent by them. Of course, parsing, syntax checking, and language translation services are provided. Type and specification description facilities and consistency checks are also provided.

These services are provided by a programming language called Act-1, a description system called Omega, a reasoning system called Ether, and a computer architecture and operating system called the Apiary. These components are being developed in parallel, and are compatible both semantically and syntactically.

The Omega Description System

Omega is a description language system for creating then augmenting a knowledge base. Knowledge is represented in terms of descriptions. The language system has a carefully-designed set of inference rules with which to reason from an existing knowledge base. Any particular description in a knowledge base need not be complete. Once additional partial descriptions of the concept are added to the knowledge base, they can be related to existing descriptions (using the inference rules) in such a manner that the order in which the descriptions were added to the database and the fact that that they were added separately are irrelevant. Using its inference rules, the language system can extract knowledge from a knowledge pool in response to queries from a user.

The Act-1 Programming Language

Act-1 is a computer programming language based on the actor model of computation. Its language constructs have sufficient and appropriate expressive power for describing actors and their behaviors. Act-1 descriptions of actors are translated by the Act-1 language system into commands runnable on a host computer architecture.

The Apiary Computer Architecture

The *Apiary* is a computer architecture designed to support actors. It is described in [Hewitt 80]. It consists of a number of processors called *workers* connected in a topology called a *folded Cartesian hypertorus.* Each worker stores a number of actors, and performs services necessary for the support of actors, such as the creation of actors, the sending of messages (which may involve cooperation with other workers), local and global real-time garbage collection of inaccessible actors, movement of actors from one worker to another, and dynamic distribution and balancing of computational tasks among workers (making use of Locality of Reference as one criterion for actor movement). Each worker consists of one or more processors for transferal of communications and for executing the instructions

of actors, a memory space private to itself for storage of actors, and an interface for interaction with other workers. Some workers may have connections to such virtual devices as disks, printers, terminals, and networks.

The Ether Problem-Solving System

Ether is a goal-oriented actor-based reasoning system. Its metaphor is that of the scientific community [Kornfeld and Hewitt 81] in which goals are disseminated, whereupon independent actors called *sprites* attempt to achieve the goals or prove that the goals can not be achieved. In working on a goal, a sprite may disemminate other goals, and may use information disemminated by other sprites. It is believed that the plurality of opinion supported by such a system will be an important part of problem-solving systems. Specific uses of *Ether* in the Act-1 language system include problem-solving by language translators and the run-time system, making use of the descriptions of actors and their relationships stored in *Omega* knowledge bases.

A User Interface

The interface to the Act-1 language system will be interpretive. A command loop will interface with input and display devices, with the *Omega* and *Ether* systems, with *Act-1* translators, and with the Act-1 run-time system. The run-time system will interface to the *Apiary* for computing power and storage for actors.

The command interpreter will accept input such as:

~ An *Omega* partial description of a concept. This includes specifications of actors and of relationships between actors.

~ An implementation of a general kind of actor, such as a Stack. This is an Act-1 description of the behavior with which to create new computational actors.

~ An operational command. That is, sending a communication to some

actor in order to begin a computation.

A special actor named `User` represents the user of the system. This actor is never garbage-collected. Its behavior knows of such things as dictionaries, knowledge bases, and any other persistent information.

Other Environments for Act-1

Although Act-1 is most powerful and useful when embedded in the language system described above, it is not the case that Act-1 must be implemented in this environment or that it can not be efficiently run on a sequential machine. Act-1 implementations for single sequential machines can be optimized greatly, communicating through shared memory and using conventional methods of control transfer. Implemented on a sequential machine, `Act-1` can be made to have performance comparable to that of sequential languages.

Chapter Four

Omega: A Description Language

4.1 Perspective

Much of the art of computer programming is really the art of description. A computer language is a description language, which can be read both by humans and by computers, for describing computational entities, relationships between them, and activities performed by them. A computer reads descriptions in the language through a translator which derives from the description a new one in a form suitable for running on the underlying computer architecture. A computer program describes new procedural or data abstractions in terms of other abstractions, which are either primitive or have already been described. A pool of such abstractions comprises a system of descriptions.

Such pools of abstractions can grow quite large and complex. Effective use by programmers of abstractions already designed and tested is often limited by a lack of structuring, cataloging, and easy access to them. Such pools are data bases in their own right, and data base management techniques can be applied to them.

An important observation is that much of programming consists of describing new abstractions in terms of old ones, and of changing descriptions. Very often, a new abstraction is very similar to an old one, or is really a special kind of an old one. The idea of being able to describe an abstraction as being basically another, but specialized in certain ways, is generally a very powerful description method, and is called *inheritance.* The general problem of description, both in natural languages and in computer-understandable languages, is one which has received much

consideration and one which is the subject of much current research. Because a programming language contains many aspects of a description language, much burden on programmers can be relieved by designing powerful description techniques into the languages which they will use. These knowledge representation and manipulation techniques blend nicely with the persistent data management techniques mentioned above.

4.2 The Omega Description Language

4.2.1 The Basics

Omega is a description system for general knowledge representation and retrieval. It is used by Act-1 to describe and reason about characterizations of actors' behaviors. It is non-trivial to choose a single word which characterizes exactly what Omega describes. Are they bits of knowledge, objects, characterizations, descriptions, categories, abstractions, concepts? We will use the term *concept* to refer to notions such as `Integer`.

Important features of *Omega* include the use of inheritance as a fundamental description method, the support of multiple partial or incomplete descriptions of any object or concept, and the support of free variables and pattern-matching for generalization of descriptions [Hewitt, Attardi and Simi 80]. In-depth discussion of *Omega* and its mechanisms is beyond the scope of this document. An understanding of the basic concepts and knowledge of the existence of *Omega*'s axioms of inference is sufficient for understanding its use in *Act-1*.

Objects or concepts in Omega are characterized by their descriptions. Instances of concepts can be described using instance descriptions. Instance

descriptions may have *attributions.* These resemble property/value or relationship/value pairs reminiscent of property lists of Lisp, and fields in records of algebraic languages. An attribute consists of a relationship and a pattern characterizing the set of values which can correspond to the relationship. For example, the concept of a complex number usually involves attributes such as real and imaginary parts, or magnitude and angle. A complex number, `3+4•i`, can be known as "a `ComplexNumber` with `realPart` 3 and with `imaginaryPart` 4". Acceptable partial descriptions of the object, `3+4•i`, include:

"a ComplexNumber"
"a ComplexNumber with realPart 3"
"a ComplexNumber with imaginaryPart 4"
"a ComplexNumber with magnitude 5"

Descriptions in *Omega* are expressed in a simple syntax resembling English language descriptions. A standard meaning is assigned to some words such as `with`, `a`, `an`, `is`, `and` and `of` in order to avoid some of the ambiguities found in the English language, and in order to impose some uniformity on descriptions. Phrases are enclosed in parentheses in order to avoid another common problem in English, ambiguity in sentence structure. Note the similarity between the English description,

"a Complex Number with real part 3 and with imaginary part 4"

and the corresponding *Omega* description,

```
(a ComplexNumber
   (with realPart 3)
   (with imaginaryPart 4))
```

What Keywords Does Omega Understand?

Omega understands a number of keywords. The keywords, `a`, `an`, and `same` indicate an inheritance relation between descriptions. The keywords, `with`, `of`, and `withUnique`, point indicate an attribution in an instance description. The keywords, `and` and `or`, are for combining descriptions (i.e. conjunction and disjunction of

descriptions). The keyword, **not**, indicates all concepts and instances *not* described by a description. Omega understands logical operators, such as $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftarrow$, and $\Leftrightarrow$, are for combining *Omega* statements. Omega understands the concepts of numbers and of relational operators, such as $<$, $>$, $=$, $\geq$, $\leq$, and $\neq$. Omega also understands a description construct for describing the behavior of an actor in terms of the communications it accepts and the communications it sends.

4.3 Inheritance Statements

Limitations or refinements may be place on a concept or object by enumerating restrictions on attributes which instance descriptions of the concept must have. This is done with the inheritance relation, **is**. For example, the fact that any particular complex number is a number means that complex numbers inherit all attributes which numbers are declared to have. In addition, any statements which are true about numbers are true about complex numbers. This inheritance is expressed as follows:

```
((a ComplexNumber) is (a Number))
```

Each statement made about a concept or object is absorbed into a knowledge base maintained by *Omega*. Partial descriptions may be entered into the knowledge base in any order.

Queries made to *Omega* are typically simple yes/no questions. In response to queries about concepts in the knowledge base, *Omega* applies a set of inference rules to related statements in order to attempt to deduce an answer.

It is often useful to describe a concept in terms of itself, to place restrictions on the attributes which instances of the concept may have. For example, all Cartesian Complex Numbers must have two attributes, a real part and an imaginary part. This

can be expressed as:

```
((a CartesianComplexNumber) is
 (a CartesianComplexNumber
    (with realPart (a RealNumber))
    (with imaginaryPart (a RealNumber))))
```

Similarly, all Polar Complex Numbers have two attributes, an angle and a magnitude:

```
((a PolarComplexNumber) is
 (a PolarComplexNumber
    (with angle (an Angle))
    (with magnitude (a RealNumber))))
```

A Complex Number in our sample system is both a Cartesian Complex Number and a Polar Complex Number:

```
((a ComplexNumber) is (a CartesianComplexNumber))
((a ComplexNumber) is (a PolarComplexNumber))
```

It is a description such as the one immediately above which allows one to speak of Complex Numbers with real and imaginary parts, magnitudes and angles.

4.3.1 Properties of the Inheritance Relation

Non-Reflexivity

The inheritance relation is non-reflexive. That is, the description

```
((a RealNumber) is (a ComplexNumber))
```

does not imply that

```
((a ComplexNumber) is (a RealNumber))
```

Transitivity

The inheritance relation is transitive. That is, if *Omega* has been told that

```
((an Integer) is (a RealNumber))
```

and that

```
((a RealNumber) is (a ComplexNumber))
```

it will be able to conclude that

```
((an Integer) is (a ComplexNumber))
```

without being explicitly told that fact.

Mutual Inheritance

A reflexive inheritance relation, `same`, is defined for convenience. It is short-hand for a pair of `is` relationships. For example,

```
((a RealNumber) same
 (a ComplexNumber (with imaginaryPart 0)))
```

means the same thing as the pair of statements,

```
((a RealNumber) is
 (a ComplexNumber (with imaginaryPart 0)))
```

and

```
((a ComplexNumber (with imaginaryPart 0)) is
 (a RealNumber))
```

4.4 Some Axioms used by Omega

This section is meant to provide a feel for the inference powers of the *Omega* system and for the mechanisms which provide this power. It is not meant to be read for in-depth understanding of these mechanisms.

Omission

The Axiom of Omission of Attributions allows *Omega* to selectively ignore attributes of an instance description. For example, the *Omega* system can make conclusions such as

```
((a ComplexNumber (with realPart 0)) is (a ComplexNumber))
```

Commutativity

The Axiom of Commutativity of Attributions allows *Omega* to permute the ordering of attributions in an instance description. For example, it allows the system to conclude that

```
((a ComplexNumber
    (with realPart 3)
    (with imaginaryPart 4))
 is
 (a ComplexNumber
    (with imaginaryPart 4)
    (with realPart 3)))
```

Extensionality

The logical operator, ⇒, is powerful when used because it can let *Omega* to deduce inheritance relations from implications. For example, knowing that

```
((=x is (a ComplexNumber)) ⇒ (=x is (a Number)))
```

allows *Omega* to conclude, from the axiom of Extensionality, that

```
((a ComplexNumber) is (a Number))
```

Three Kinds of Attributions

Some attributions are weaker than others. That is, more powerful inference rules can be applied to some attributions and not to others. Three useful kinds of attributions have been found. Of these, the `of` attribution is the weakest. The `of` attributions of instance descriptions can only be manipulated with the basic and very general axioms. A typical use would be for binding actual parameters to formal parameters in a procedure call.

The `with` attribute is stronger, allowing instance descriptions of a concept to be Merged. For example, if something is

```
(and (a ComplexNumber (with realPart 5))
     (a ComplexNumber (with imaginaryPart 3)))
```

then *Omega* can merge the partial descriptions to conclude that it is

```
(a ComplexNumber
   (with realPart 5)
   (with imaginaryPart 3))
```

The `withUnique` attribute is strongest attribution because it makes

assumptions about the values which must appear in an attribution. For example, suppose Omega knows, presumably by having merged two partial descriptions, that

```
(=x is
 (a ComplexNumber
    (withUnique realPart (an Integer))
    (with realPart y)))
```

From this, it can conclude that

```
(=x is
 (a ComplexNumber
    (withUnique realPart (and y (an Integer)))))
```

Other Axioms

Omega has other axioms and mechanisms which are beyond the scope of this document. Interested readers should read [Hewitt, Attardi and Simi 80] and [Attardi and Simi 81].

4.5 Pattern-Matching Facilities

Simple Patterns

Pattern-matching is often useful for describing instances of a concept. Typically, a pattern is simply a description of a concept which places restrictions on the attributions of instances of the concept. For example,

```
(a ComplexNumber
   (with realPart (> 0))
   (with imaginaryPart (< 5)))
```

describes an infinite set of Complex Numbers, in which `3 + 4•i` is included, but `3 + 7•i` is not.

The pattern,

```
(a ComplexNumber
   (with realPart 3)
   (with imaginaryPart 4))
```

is much more restrictive, because it describes only the complex number, `3 + 4•1`.

The pattern,

```
(a ComplexNumber
   (with realPart (a RealNumber))
   (with imaginaryPart (a RealNumber)))
```

is much more general, describing all complex numbers.

Patterns With Constraints

When a pattern-match is performed, it is often useful to bind parts of the matching instance description to identifiers, to allow those parts to be referenced elsewhere in a description. A very common use of pattern-matching with identifier binding is for characterizing the communications which an actor will accept, and the events it will cause if it accepts each. Each behavior has a set of communication handlers. Each communication handler consists of a pattern for communications and a body of commands to be executed if a communication matching that pattern is received. The body of commands usually makes use of information (in the form of actors) contained in the communication. This is achieved by binding identifiers in the pattern to those actors which are to be used, and by referring to these actors using the identifiers bound in the pattern.

For example, a `Request` is defined to be a communication, and to have two attributes, a message and a customer. Its specification is quite simple:

```
((a Request) is
 (and
     (a Communication)
     (a Request
        (with message (a Message))
        (with customer (a Customer)))))
```

If Act-1 commands in a communication handler wish to reference the request itself, they may use a pattern in which an identifier (e.g. `r`) is bound to the actor by the binding operator, `=`. The qualifier, `WhichIs`, is used to place restrictions on the

actors which can be bound to the identifier. In the following pattern, only actors which are Requests will match the pattern, and the identifier, `r` will be bound to any matching request:

```
(=r WhichIs (a Request))
```

The body of commands usually does not care about having a handle on the request itself, but cares about referring to the message and customer in the request. If the body of commands wishes to accept any Request, the pattern does not need to put restrictions on the customer or message[3]:

```
(a Request
   (with message =m)
   (with customer =c))
```

A pattern can go into arbitrary detail, such as a pattern which might be used for a push requisition in a stack of Integers:

```
(a Request
   (with message
       (a PushRequisition
          (with item (=i WhichIs (an Integer)))))
   (with customer =c))
```

Interesting relationships between concepts can be characterized using these pattern-matching techniques. For example, the mapping from Polar to Cartesian Complex Numbers can be represented in *Omega* by showing the mapping for one general (pattern for a) Polar Complex Number to a corresponding Cartesian Complex Number:

```
((a PolarComplexNumber
    (with angle =theta)
    (with magnitude =rho))
 is
 (a CartesianComplexNumber
    (with realPart
        (* rho (a Sine (of angle theta)))
    (with imaginaryPart
        (* rho (a Cosine (of angle theta)))))))
```

---

[3] `m` is restricted to be a `Message`, and `c`, a `Customer`, by the definition of *Request*.

4.6 The Use of Omega in Act-1

Because of the need for descriptive power in a computer language, *Omega* permeates *Act-1*. It is a bit early to go into much detail, but an overview is useful for understanding. Actual use of *Omega* will be quite apparent in *Act-1* code as the description of the language proceeds. Omega is used for specifying properties of actors, in a manner which is reminiscent of type declaration in many familiar languages. Because of the power in Omega's basic description mechanisms, much information can be declared about each actor, and a more comprehensive form of consistency checking can be performed.

Act-1 can use inheritance relations between concepts described in Omega for checking interface constraints. For example, if a pattern calls for a `ComplexNumber`, and the `RealNumber` 4.3 is being checked, 4.3 will be thought of as `4.3 + 0*i` and used that way, as long as Omega can conclude from its system of descriptions that

```
((a RealNumber) is
 (a ComplexNumber (with imaginaryPart 0)))
```

Its task is even simpler if it contains a more direct representation of the relationship:

```
((=r WhichIs (a RealNumber)) is
 (a ComplexNumber
    (with realPart r)
    (with imaginaryPart 0)))
```

Pattern-matching is an important feature in Act-1. Decisions in specifications and implementations are often made by determining whether or not an actor's behavior matches a specified pattern.

In an actor's implementation, assertions, in terms of patterns representing actors from a communication or actors known to an actor, can be made about that actor's behavior or about the behavior of other actors involved in its implementation. Such assertions serve as comments to the human reader and are

machine-readable as well.

Often, there are actors with very simple behaviors which correspond to record structures in many familiar languages. Omega can implement simple behaviors such as those corresponding to record structures which do not update their components. An Omega description naming its attributes and their contents is sufficient to create such an actor. A simple Omega pattern-match can bind identifiers to the contents of the actor in order to reference them.

Chapter Five

An Overview of the Language

5.1 Motivation for the Language Design

The design of the Act-1 language system was largely motivated by the inadequacy of existing language systems and by the unexploited developments in hardware technology. In his ACM Turing Award paper [Backus 78], John Backus has eloquently pointed out many of the inadequacies of existing algebraic languages. Inadequacy of existing languages comes largely from inadequacies in their computational models.

5.1.1 Design goals of the Act-1 Language

The Act-1 language system was designed to fill a need for a language system which could exploit advances in computer hardware and software engineering. Hardware advances have made concurrency a must in computer languages. Software engineering advances have taught us the benefits of encapsulation of detail, the benefits of absolute containment of implementation details. Work in verification of software abstractions has taught us that economy of mechanism is desirable in a language.

The goals of Act-1 include the following:

~ Implement the Actor computation model, which supports concurrency of activity and absolute containment.

~ Have economy of mechanism, to make the language kernel small, to make learning the language easier, to make proving program properties

easier, and to make its implementation simpler.

~ Have conceptual unity of language constructs. In particular, unify procedural and data abstractions, and, correspondingly, integrate behavioral descriptions and operational commands.

~ Expressiveness. Make expression of abstractions easy for programmers by providing simple, but powerful primitives and abstraction mechanisms, convenient abbreviations, and some amount of programming automation.

~ Make abstractions be expressed in a form in which they can be reasoned about by humans and programs.

It is worthwhile to point out the difference between expressive power and expressiveness. Expressive power is related to the notion of a set of all algorithms which can theoretically be expressed in a language. The Turing machine is often used as a measure of expressive power. Anything which can be done with a conventional language can theoretically be done with a Turing machine, but programming it would be no picnic!

5.1.2 Assumptions About the Underlying Act-1 Run-Time System

The following are some assumptions about the environment in which actors are to exist (that is, in which Act-1 programs are to be run). These assumptions influence the style in which Act-1 programs are written. The reader is urged to view the language, its constructs, and its intended programming strategies in light of the following assumptions:

~ Creation of actors is very cheap.

~ The garbage-collection algorithm for reclaiming the storage of inaccessible actors, and is effective and efficient. It works in real time, recovering temporary storage quickly.

~ Delivery of communications is relatively cheap, quick, and very reliable.

~ Memory is inexpensive and plentiful.

~ Copying and maintaining multiple copies of actors on different processors is cheap (storage is quickly reclaimed by a garbage collector).

~ The underlying computer architecture may consist of large numbers of interconnected processors.

Some of the above assumptions may not seem plausible at first glance. Bear in mind that what may not seem intuitively efficient to a programmer who is used to conventional languages and von Neumann machines may have overall efficiency gains, or be a relatively small loss, in decentralized environments. In addition, there is much potential for optimizations to be made by the language system. The garbage collection and load-balancing systems can cooperate to maximize locality of reference. The sending of a communication from one actor to another actor residing on the same worker can be optimized to have an overhead which is on the order of a procedure invocation in conventional languages. Because of the principle of locality of reference, it is expected that a large fraction of all communications sent by actors will be to target actors which are on the same worker as the sender. The time-critical software in the implementation of the garbage collection and mailing systems will be written in micro-code, or implemented in hardware, in order to decrease its running time.

In any case, the reader should bear in mind that costs are never absolute, but must be weighed against benefits and alternatives. There are many benefits nested in *Act-1*, including potentials for concurrency, powerful description primitives, and abstraction from the programmer of details such as memory management, transmission of communications, and generation of parallel activity. We believe these software benefits are worth extra hardware cost.

5.2 An Overview of the Act-1 Language

This section is intended to give the reader a very general or intuitive feel for the language and its use. It displays some uses of a few language features. It attempts neither to explain the full meaning of the features nor to display all of the features and their usage. More detailed descriptions of language features follow, which are intended to clarify semantic and syntactic questions.

Recall that in the actor model, computations proceed as each actor receives and processes *communications*. *Communications* are just actors which are constructed to contain information, then to be sent to some actor. Three basic types of communications exist. An actor may wish to send another actor a *Request* to do some processing involving specified actors and to respond to a specified customer. An actor processing a request for some customer may successfully complete the request, and send a `Reply` to the customer. An actor processing a request for some customer may not be able to successfully complete the request, and send a `Complaint` to the customer.

For example, some actor wishing to print the factorial of some number, `n`, might send the `Factorial` actor a *Request* that the factorial of `n` be sent to some customer which accepts the response from factorial and prints the results. The `Factorial` actor might accept a request with an integer and a customer and proceed as follows. If the integer is less than zero, `Factorial` complains to the customer in a `Complaint`; otherwise, it computes the factorial of the number then sends the result to the customer in a `Reply`.

When an actor receives a communication, it performs some computation based on its interpretation of the communication. Intuitively, an actor's set of "stimulus-response patterns" is known as its *behavior*. In *Act-1*, a description of an

actor's behavior consists of a characterization of the communications it will accept and of the actions it will perform as a result of receiving each communication. The kinds of actions an actor can perform are: creating new actors, sending communications to actors, and replacing its current behavior. Communications are characterized using the pattern-matching facilities of *Omega.*

Act-1 syntax for describing the implementation of a behavior (e.g. that of an actor named **SumFromZero**) is[4]

```
(SumFromZero be
  (new Behavior
     CommunicationHandler
     ...
     CommunicationHandler))
```

Each communication handler characterizes, using pattern-matching, one kind of communication which the actor will accept. Using names bound in the pattern match, each handler also characterizes the processing which will occur due to the acceptance of a communication of that type. A communication handler consists of a pattern, a keyword indicating the nature of the pattern, and a body of commands to be executed if the pattern matches a communication received. Each communication handler describes what the actor will do to process a communication which matches a particular pattern. The set of all communication handlers defined for an actor comprises the actor's behavior. The syntax for a general communication handler is

(is *CommunicationPattern* **communication** *body***)**

Handling a Request

One type of communication is called a Request. It contains two pieces of information, a message for the receiver of the communication, and a customer to

---

[4]Note the use of the imperative construct, **be**, in implementations, as opposed to the descriptive construct, **is**.

which the receiver should send a response. By convention, a message in a Request is called a Requisition. It can contain any attributions with information for the receiver of the Request. The sender should only send requisitions whose attributes the receiver is prepared to handle[5]. For example, an actor which calculates the sum of all integers from 0 to some positive upper bound might be written to respond to requests such as:

```
(a Request
   (with message
      (a Requisition (with upperBound 10)))
   (with customer c))
```

One possible implementation of `SumFromZero` might be one which receives a request like the one above and computes the sum using the closed form for the summation, `N * (N-1) / 2`. It then sends the result to the customer, `c`, of the request. The implementation below only accepts Requisitions in which the upper bound is a NonNegativeInteger.

```
(SumFromZero be
  ; comments begin with a semicolon and end with a newline.
  (new Behavior
    ; The following is called a communication-handler.
    (is (a Request
      (with message
        (a Requisition
         (with upperBound
               (=n WhichIs (a NonNegativeInteger)))))
      (with customer =c))
     communication
         ; when a communication of the above form
         ; above is received, the following is done.
       (SendTo c
         (a Reply
            (with message (/ (* n (- n 1)) 2)))))))))
```

The following is a different and slightly more complicated implementation of `SumFromZero`, which illustrates a multi-way decision and the generation of

---

[5]Otherwise, the Act-1 system will generate a complaint.

complaints. The actor accepts any integer as an upper bound, and complains when the upper bound is less than zero.

```
(SumFromZero be
  (new Behavior
    (is (a Request
      (with message
        (a Requisition
          (with upperBound (=n WhichIs (an Integer)))))
      (with customer =c))
     communication
      (CaseFor n
        (is (< 0) then
          (SendTo c
            (a Complaint
              (with message negativeUpperBound))))
        (is 0 then
          (SendTo c (a Reply (with message 0))))
        (is (> 0) then
          (SendTo c
            (a Reply
              (with message (/ (* n (- n 1)) 2)))))))))))
```

Act-1 provides very fundamental and very expressive primitives. As a result, programmers have great latitude with respect to how they can express an implementation. The primitive methods of expression tend be more verbose and to describe an actor's activities at a low, message-passing level. Syntactic sugar and natural extensibility in the language remove much of the cumbersome aspects of the more primitive methods. More declarative implementation methods can be used to avoid much of the low-level detail. Ultimately, though, such declarative descriptions are transformed into the more primitive forms by Act-1 translators.

At the descriptive extreme, `SumFromZero` could be implemented as follows:

```
((new SumFromZero
    (with upperBound (=n WhichIs (a NonNegativeInteger))))
 be
 (/ (* n (- n 1)) 2))
```

Chapter Six

Transmission of Information Between Actors

6.1 Perspective

Communication between actors must be generalized to abstract away the location of actors, since actors can be moved from worker[6] to worker in a manner invisible to the programmer. Actors must be able to transmit information to other actors, which may be on the same worker or may be on some other worker somewhere on a packet-switching network.

There exist many issues which must be dealt with in packet-switching networks, and many alternatives with which to deal with them [Kleinrock 76, Pouzin and Zimmerman 78]. Information to be transmitted is packaged in one or more packets and sent from one worker toward another. If the source and destination workers are not adjacent in the network topology, then the packets are passed from worker to worker until the destination worker is reached. The problem of choosing one of many paths from the source worker to the destination worker is called *routing*. Conversation between workers can take arbitrary lengths of time, leading to arbitrary delays in transmission of information. If a communication is composed of several packets, they may arrive at the destination worker out of order. Some packets can be lost or duplicated. The contents of packets may be damaged during transmission. The processing speed of a worker may not be the same as that of those sending it information. Each and every one of these problems must be dealt with at

---

[6]Recall that a *worker* is one of a number of cooperating processors in a computer architecture. Each worker provides storage and processing power for some actors.

some level in the language system or in its support system.

At some level, the system must be able to deal with fragmentation of information to be transmitted into packets, and reassembly of packets into information. At some point buffering may be used to help solve the problems of reassembly and speed mismatch. Flow control mechanisms can also help deal with speed mismatch. Routing is necessary when source and destination workers are not adjacent in the network topology. Sequencing and numbering strategies may be used to help deal with missing, duplicate, or reordered packets. Checksums help recognize and deal with damaged packets.

Above this layer of problems lies the need for choosing among pairs of alternatives. Should communication be synchronous or asynchronous? Transmission of information between workers may be synchronous, requiring the attention and cooperation of source and destination workers (and possibly actors), or asynchronous, allowing destination actors not to be aware of the transmission process. Should communications be buffered or unbuffered? If an actor is busy when transmission of a communication to it is attempted, the transmission can be made to fail, or the communication can be stored in a buffer from which the actor will take communications when it is ready to. Should processing of communications be fair? Does an actor process communications in the order in which they were transmitted?

6.2 The Mailing System Abstraction

In Act-1, transmission of communications is fair, asynchronous, and buffered. The Act-1 abstracts away from the programmer all aspects of the process of transmitting information between actors. There exist operations in the language for mailing communications to actors. The underlying system deals with the problems incurred in the transmission, such as the peculiarities of packet-switching networks.

The abstraction of the mailing system provides a simple interface for the programmer using the transmission operations. Act-1 code need only designate an actor to be sent, and invoke a sending operation, indicating the actor to which it should be sent. The Act-1 programmer can assume that the transmission will be a success. If problems in transmission arise, they will surface through the language's exception-handling mechanisms, and be handled at an appropriate level.

6.2.1 Packaging of Information for Transmission

Information to be transmitted from one actor to another is first packaged in a third actor, called a *Communication.* In the general case, the contents of a Communication is arbitrary. It is, of course, useful for both the sender and receiver to be able to understand the format of the Communication. In Act-1's own terms,

```
((a Communication) is (an Actor))
```

There are three common protocols for sending information to an actor. These correspond to three natural communication modes in message-passing semantics, requesting that an actor do something, normal response to a communication with a reply, or exceptional response to a communication with a complaint. A convenient kind of Communication actor has been defined for packaging the information involved in each.

One protocol for sending information to an actor is to request that it process some particular information and to designate the actor to which it should acknowledge its processing and send the results of its processing. A special kind of Communication called a *Request* exists for containing these two pieces of information. All requests have a *customer* attribute, which designates the actor to whom the processor of the Request should respond. All requests also have a *message* attribute. If the target actor is capable of performing various kinds of processing, the message indicates which kind is desired. If the target actor needs any additional information to perform the processing, the message also designates that information. Any actor can serve as a *message* attribute. It is desirable for both the sender and receiver to agree on the interpretation of messages.

In Act-1 terminology,

```
((a Request) is
 (a Request
    (with message (a Message))
    (with customer (a Customer))))
```

Another common reason for sending information to an actor is to acknowledge the completion of the processing of a request. Such an acknowledgement often contains information resulting from the processing of the request. Two kinds of acknowledgements are useful.

One, called a *Reply*, acknowledges successful completion of the processing of a request. All replies have a *message* attribute designating relevant results of processing or indicating its completion.

```
((a Reply) is
 (a Reply
    (with message (a Message))))
```

Another, called a *Complaint*, acknowledges an exceptional termination of the processing of a request. All complaints have a *message* attribute indicating the

reason for the exceptional processing of the request and, possibly, designating useful information about the processing, such as intermediate or alternative results.

```
((a Complaint) is
 (a Complaint
    (with message (a Message))))
```

6.2.2 Transmission of Communications

The Act-1 language contains operations for transmitting communications to actors. All of these operations enable the programmer to assume that the communications will reach their destination. The Act-1 mailing system ensures transmission of communications. Problems it cannot deal with are dealt with using mechanisms which are beyond the scope of this document.

6.2.2.1 SendTo

The most general operation for sending communications is called *SendTo*. In fact, all other communication operations are derived either directly from *SendTo*, or with some syntactic sugaring. Its invocation must designate both the communication to be transmitted and the actor to which it is to be sent. The form of this command is

(SendTo *targetActor communication*)

The following implementation of an actor named Negate contains an example of the *SendTo* command. The portions of interest are underlined. The rest of the implementation provides context and a feel for the structure of the language.

```
(Negate be
  (new Behavior
     (is (a Request
          (with message
             (a Requisition
                (with number (=n WhichIs (an Integer)))))
          (with customer =c))
         communication
          (SendTo c
             (a Reply (with message (- 0 n)))))))
```

6.2.2.2 Abbreviations of SendTo

Useful operations exist for sending each of the three common kinds of communications. The *ReplyTo* operation transmits a Reply. The *ComplainTo* operation transmits a Complaint. The *Ask* expression is a syntactic sugaring which facilitates the expression of two-way communication. It appears in Act-1 code as it transmits a Request to some target actor then waits for the target actor's response. The target actor's response is the value of the Ask expression.

The *ReplyTo* and *ComplainTo* operations are merely convenient abbreviations of the *SendTo* operation. The two operations are very similar in function.

6.2.2.3 ReplyTo

The *ReplyTo* operation is invoked with an indication of the destination actor, in a manner similar to the invocation of *SendTo*. Instead of designating a communication, however, only the message to be sent in a Reply is designated. The *ReplyTo* operation creates a Reply containing the message, then sends that communication to the destination actor, in the same way *SendTo* does.

The command, `(ReplyTo` *targetActor message*`)`, is exactly equivalent to the command,

```
(SendTo targetActor
  (a Reply (with message message)))
```

The following implementation of the Negate actor is exactly equivalent to the one above. The only textual difference is that this one uses *ReplyTo*, whereas the former used *SendTo*.

```
(Negate be
  (new Behavior
     (is (a Request
         (with message
            (a Requisition
               (with number (=n WhichIs (an Integer)))))
         (with customer =c))
        communication
         (ReplyTo c (- 0 n))))))
```

6.2.2.4 ComplainTo

Similarly, the *ComplainTo* operation is a simple abbreviation of the *SendTo* operation. The command, `(ComplainTo <targetActor> <message>)`, is entirely equivalent to the command,

```
(SendTo targetActor
   (a Complaint (with message message)))
```

For example, consider an actor, `MultiplicativeInverse`, which computes the multiplicative inverse of some Real Number sent to it. This actor would need to complain if the number, `0` were sent to it.

```
(MultiplicativeInverse be
 (new Behavior
      (is (a Request
          (with message
              (a Requisition
                 (with number (=r WhichIs (a RealNumber)))))
          (with customer =c))
        communication
        (if (n is 0)
            then (ComplainTo c (a DivideByZeroComplaint))
            else (ReplyTo c (/ 1 r))))))
```

6.2.2.5 Ask

The operation dealing with Requests is slightly more complicated to understand, but is equally simple to use. The *Ask* operation is treated like an expression which, given a target actor and a message, will send the message to the target actor in a Request, then await the response from the target actor, which it will yield as the value of the expression. The form for the *Ask* expression is

(Ask *targetActor message*)

A trivial example of the use of the *Ask* expression is in the following implementation of an actor called MinusFive.

When MinusFive is sent a Request with any Requisition as its message, it will ask Negate to negate the integer 6. When it receives the Reply from Negate, it will add 1 to the value in the reply, then mail the resulting value, -5, to its original customer in a Reply.

```
(MinusFive be
  (new Behavior
     (is (a Request
          (with message (a Requisition))
          (with customer =c))
        communication
         (ReplyTo c
           (+ 1
             (Ask Negate
               (a Requisition
                  (with number 6)))))))
```

Two-way conversation between actors is implemented by Act-1 using the standard one-way communications. Recall from page 13 the idea of pipelining the activities of actors. The context in which the Ask expression is enclosed represents what will be done once the response is received via the Ask expression. This is a behavior and can be performed by an actor. Equivalent to having an Ask expression is the idea of sending a Request to the target actor, containing the Message for the

target, and indicating as the customer of the Request a special *customer* actor.[7] The *customer* is dynamically created just before the Request is sent. Its behavior is such that the customer will accept the response from the target actor, then continue processing. Using this perspective, another implementation can be written for the actor, MinusFive, without using the *Ask* expression.

```
(MinusFive be
 (new Behavior
  (is (a Request
    (with message (a Requisition))
    (with customer =c))
   communication
   (SendTo Negate
      (a Request
       (with message
          (a Requisition
           (with number 6)))
           (with customer
              (a Customer
               (with behavior
                  (new Behavior
                   (is (a Reply
                     (with message
                        (=n WhichIs (an Integer))))
                     communication
                     (ReplyTo c (+ 1 n)))))))))))
```

6.3 Summary

The basic communication primitive is the *SendTo* command, which reliably sends a communication to some destination actor. The actor using *SendTo* needs not wait for the sending to complete, because it is buffered and assumed to be reliable. The syntax for a *SendTo* is

(SendTo *actor communication*)

Three basic communications are defined, for convenience and convention.

---

[7]Comments on the relationships between customers and continuations can be found in [Hewitt and Attardi 81].

Each has a standard sending operation which is based on *SendTo*, but which is more convenient to use to send a communication of that particular type.

An actor usually responds to some customer as a result of accepting a communication. A normal response to a complaint is usually done by sending a reply containing some message for the customer. That is,

```
(a Reply (with message message))
```

Such a *Reply* can be sent to a customer by using the *ReplyTo* transmission command,

```
(ReplyTo customer message)
```

An alternative response to a communication is an exceptional one, in which an explanatory message is sent to a customer in a *Complaint* communication:

```
(a Complaint (with message message))
```

Such a *Complaint* can be sent to a customer by using the *ComplainTo* transmission command,

```
(ComplainTo customer message)
```

The third basic communication is a *Request* sent to an actor asking that it perform some computation using the information in some designated message and respond to some specified customer. A *Request* has the form,

```
(a Request
   (with message message)
   (with customer customer))
```

The *Ask* expression,

```
(Ask target message)
```

can be embedded in other expressions or commands. In the code, it appears as if this expression sends *message* to the target actor, *target*, then yields the actor's response as its value. This is, however, just a syntactic sugaring.

Chapter Seven

Description of Behavior

7.1 Perspective

7.1.1 What is Behavior in Act-1?

Recall that computation in the actor model occurs as actors accept communications and, based on the communications they accept, they

transmit more communications,
create new actors,
and change the way they react to further communications.

The way an actor reacts to communications (intuitively, its "stimulus-response patterns") is known as its ***behavior.***

The process of describing an actor is really the process of describing the actor's behavior. Such a description is a characterization of the communications the actor will accept[8], and a characterization of what is done when each communication is accepted.

What can an Actor Do?

An actor can create new actors. An actor can send communications to actors. An actor can designate a replacement behavior for itself, which is installed before the next communication is accepted. A few special actors are implemented in hardware or microcode. They perform special computational tasks in addition to

---

[8]Communications sent to an actor, but rejected by the actor, are in general handled by sending a complaint to the customer.

those mentioned above.

7.1.2 Two Important Views of an Actor

There are two major perspectives from which to view an actor. Human users and other actors view an actor and see ***what*** it does in response to various communications. The programmers who define an actor and the computer architecture which supports its activities see ***how*** the actor does what is seen from the user's perspective. The two views, a user's and an implementor's, are distinct, but related in important ways.

Specification of Behavior

The description of an actor from the user's view is typically called a *specification*. A specification can be as little as a few human-oriented comments or as much as a formal, machine-readable definition of the actor's behavior, from which a naive implementation could be generated with little effort. Typically, a specification will contain descriptions of or assertions about the properties of an actor and about its relationships with other actors. A specification of an actor is useful because it allows interested parties to make use of the actor without knowing all the details involved in machine-runnable descriptions of the actor, and because it allows interested parties to reason abstractly about actors and systems of actors.

The specification of an actor in the Act-1 language system consists of descriptive information about the actor, expressed in the *Omega* language. Such descriptive information can be embedded in a collection of *Omega* partial descriptions. One of the beauties of *Omega* is that it allows incremental entry of partial descriptions in *Omega* is the fact that a programmer can begin by describing

very simple and general properties about an actor. Once these are digested[9] by *Omega*, the programmer can add successively more detailed descriptions of the actor. As each is pondered by *Omega*, it performs consistency checks to make sure that this description is consistent with related descriptions it has already digested. Inconsistency is a very common bug in hand-written specifications, and is naturally more likely to occur in more detailed specifications.

Implementation of a Behavior

An implementation provides a machine-readable and machine-runnable description of an actor's behavior. It determines *how* an actor really reacts to communications. To express an implementation, Act-1 uses the description facilities of Omega. In addition, it provides two powerful primitives. One indicates the creation of an actor with a specified behavior. The other indicates a replacement of the current behavior with a new one. These constructs are for instructing the underlying computer on what it must do to support the activities of actors.

Some declarative information can be embedded in Act-1 code. It is often the case that programmers make assertions in comments, which state key assumptions about the computations being performed, such as assumptions about valid ranges of values or about relationships between different actors involved in the implementation. These are typically readable by humans, but are not by the language translators. Such declarative information can be expressed in *Omega* and embedded in Act-1 code. Its use is encouraged. In fact, its use could improve the performance of programmers. It is believed that many problems in implementations of actors might be due to implicit assumptions made by a programmer about the nature of the contents of a communication, about protocols

---

[9] A similar process of digestion is described more fully by Fahlman, in the context of his Netl system [Fahlman 79].

required of a user, or about the relationships between actors in the implementation. It is believed that when an implementor consciously thinks about the assumptions being made in an implementation, with the intent of describing them with invariants, the degree to which the implementation will be plagued with bugs from unconsidered assumptions will be lower. The enforcement of such assertions or invariants may be supported by the run-time system at some time in the future, but until then, they may serve as precise comments for implementors.

An important goal in programming is to make the implementation of an actor consistent with its specification. Demonstrating such consistency can be done either formally or informally. As the Act-1 language system evolves, it will increasingly help implementors perform this task.

7.2 Description of Behavior in the Act-1 Language System

Specifications and implementations in the Act-1 language system have very similar syntax. In fact, the only portions of implementations which cannot be expressed in Omega are creation of actors and behavior transition. Because of this, we will begin by pointing out these differences, then will proceed to present what they have in common in whichever of the two contexts is most convenient at the time. The reader should have no trouble distinguishing between the two.

7.3 Omega Specification of Behavior

There are two basic ways of specifying an actor's behavior. One is to focus on the properties of the actor by characterizing its attributions. This is done using standard Omega descriptions. The other is to focus on its behavior in terms of what communications it will accept and what communications it will generate as a result

of receiving each. Omega recognizes a special form of description for this, which describes an actor in terms of its behavior, on a message-passing level. These two methods are very intimately related, and each can be thought of in terms of the other. In this chapter, we will focus on the second of these methods.

To specify the behavior of checking accounts in general, an Omega specification might be structured as follows:

```
((a CheckingAccount (with balance (a DollarValue)))
 is
 (a Behavior
   CommunicationHandler
   ...
   CommunicationHandler))
```

That is, a checking account is a special behavior which accepts various communications and does various things in response. Each communication handler characterizes some set of communications (such as the set of deposit requests and the set of withdrawal requests) and the actions the actor will take upon accepting such a communication. Communication handlers will be discussed shortly.

The key to distinguishing a specification from an implementation is that it speaks of ***a** Behavior*. The indefinite article, *a*, is always part of Omega syntax.

An important thing to remember is that a specification, like any Omega description, need not be complete. In fact, since Omega cannot implement actor creation or replacement of behavior, Omega can not give a complete description of the actor's activities.

7.4 Act-1 Implementation of a Behavior

To implement an actor in Act-1 is to describe how to create a new one. To do this is to describe its initial behavior in enough detail to enable the run-time system to create an actor which the underlying architecture can support.

Once again, there are two basic alternative ways to do this. Here, too, we will emphasize the way which implements an actor in terms of the messages it receives and sends, of the actors it creates, and of its designation of new behavior.

For example, an implementation of checking accounts in general would demonstrate the creation of a new checking account with some initial balance:

```
((new CheckingAccount
      (with balance (=b WhichIs (a DollarValue))))
 be
 (new Behavior
     CommunicationHandler
     ...
     CommunicationHandler))
```

Notice the use of the word **new** in the above illustration. This word always refers to the creation of a Behavior. An implementation can always be recognized as such because it must contain the word **new**.

The creation of a particular actor, such as **factorial**, may be done in a manner very similar manner:

```
(factorial be
 (new Behavior ...))
```

7.4.1 Dynamic Creation of Actors

Just as a communication can be sent by using the **(SendTo ...)** operation, a new **CheckingAccount** with balance 0, as defined above, can be created using the expression,

```
(new CheckingAccount (with balance 0))
```

An actor can also be created by spelling out the complete instructions for its creation. That is, by saying `(new Behavior ...)` in an appropriate context, a new actor with that behavior will be created.

7.4.2 Replacement of an Actor's Behavior

As a result of receiving a communication, an actor may change its behavior. This is done by using the *become* statement:

(become *replacementBehavior*)

This statement simply designates the behavior which is to replace the actor's current behavior. This can be done either by naming an existing actor or by dynamically creating a new actor. The actual replacement of the behavior can be thought of as occurring just before the next communication is accepted.

7.4.2.1 Constant Behavior Can Be Replicated

When creating an actor, Act-1 notices whether or not the behavior definition contains a *become* expression. If it has none, then its behavior cannot change. In that case, multiple copies can be made of it, each of which can accept and process messages concurrently with the others. Actors whose behavior cannot change are called *unserialized actors.* Actors whose behavior can change are called *serialized actors.* Because its behavior is subject to change as a result of receiving and processing a communication, a serialized actor can only accept one communication at a time.

7.5 Communication Handlers

The description of a behavior consists of one or more *communication handlers.* Each communication handler characterizes, using a pattern, some set of communications which the actor will accept and process. Each handler also contains a body of commands which describe, perhaps using identifiers bound in the pattern, the processing of any communication matching the pattern. The syntax for a communication handler is:

`(is` *patternForCommunication* `communication` *bodyOfCommands*`)`

It is within the body of a communication handler that statements such as the `SendTo` and `become` statements appear. It is in that context in which actors are dynamically created using the `new` expression.

Examples of Communication Handlers

We will illustrate the use of communication handlers within a specification. A simple specification is that of checking accounts mentioned above. Our specification will enumerate the communications a `CheckingAccount` can receive, and the communications it might send in reaction to each. Each of our CheckingAccount actors will (by the conventions we establish when we specify Checking Accounts) accept three basic kinds of requests. One might be a Request containing a Requisition for the current balance. In response to such a Request, a CheckingAccount actor would send a Reply to the specified customer, containing the current balance. One might be a Request containing a Requisition to deposit a specified amount of money. To this, a Checking account actor would reply with a report indicating completion. Another might be a Request containing a Requisition to withdraw a specified amount of money. To this, a Checking Account actor might respond with a Reply indicating completion or with a Complaint indicating that the balance is too small. Notice that our specification says nothing about when and how

changes to the current balance of a CheckingAccount happen. It leaves that to an implementation. Here is a specification:

```
((a CheckingAccount (with balance (a DollarValue))) is
 (a Behavior
  (is (a Request (with customer =c)
                 (with message (a BalanceRequisition)))
      communication
      (ReplyTo c (a DollarValue)))
  (is (a Request (with customer =c)
                 (with message
                       (a DepositRequisition
                         (with amount (a DollarValue)))))
      communication
      (ReplyTo c (a Reply (with message Completed))))
  (is (a Request (with customer =c)
                 (with message
                       (a WithdrawRequisition
                         (with amount (a DollarValue)))))
      communication
      (or (ReplyTo c (a Reply (with message Completed)))
          (ComplainTo c (a Complaint
                            (with message Bounced)))))))
```

Notice how each communication handler handles a different set of communications which a CheckingAccount might accept.

7.6 Acceptance of a Communication for Processing

When an actor receives a communication, it must determine which, if any, of the handlers apply to the communication. If the communication matches exactly one pattern, the corresponding body will be executed. If the communication matches the patterns in more than one of the handlers, then one[10] is chosen to handle the communication. Once a handler is chosen, its body is chosen for

---

[10] The first pattern determined to be a match is chosen. "First" means temporally first. This allows an implementation in which patterns are checked concurrently. In concurrent implementations, the selection cannot be predicted by textual positioning of the pattern, so the selection appears to the programmer to be a non-deterministic choice. Because of this, patterns should in general be mutually exclusive.

processing of the communication. Identifiers in the pattern portion of the handler which are bound to corresponding portions of the communication being processed may be used in the body of the handler to express manipulations of the corresponding actors.

If the communication matches no pattern, then the Act-1 run-time system will generate an appropriate complaint to the customer of the request.

7.7 Binding Identifiers to Actors

We have seen how identifiers can be bound to portions of an actor which is matched with a pattern. Other means exist for binding identifiers to actors. This can be done globally at the command-loop level, with a statement such as `(factorial is ...)`. Two constructs exist for binding identifiers to actors in commands or expressions. All binding in Act-1 is lexically scoped.

7.7.1 *Let* Commands and Expressions

An identifier can name an actor, such as a dynamically created actor or the result of an expression, for use in another expression or in a body of commands, using the `let` statement. In the following statement, *identifier* is bound to the actor which results from the evaluation of *expression*. The binding is valid in the scope of the Act-1 code, *code*, which can either be an expression or a body of commands. The `let` statement is an expression if *code* is an expression; otherwise it is a command.

`(let (`*identifier* `be` *expression*`) in` *code*`)`

For example, the following command binds the identifier `x` to the actor `5`, and sends a Reply with message `6`.

```
(let (x be 5) in (Reply (+ x 1)))
```

7.7.2 *Label* Expressions

A self-referencing expression can be written with the `label` expression[11], which binds an identifier to the expression, whose scope is the expression itself. The following statement binds the identifier *identifier* to the Act-1 expression, *expression*:

`(label` *identifier expression*`)`

It is a bit early to provide a sensible example, so here is a trivial, but silly, one. It is a self-referencing description:

```
(label ZeroSequence
   (a Sequence
      (with first 0)
      (with rest ZeroSequence)))
```

7.8 Control Structure

7.8.1 A Body of Commands

In Act-1, a body of commands simply consists of some number of adjacent commands. By default, all of these commands are executed concurrently. A body of commands simply looks like:

command command ... command

---

[11] Note that the `label` expression,

`(label` *identifier expression*`)`

is just an abbreviation for the `let` expression,

`(let (`*identifier* `be` *expression*`) in` *identifier*`)`

7.8.2 Sequencing of Commands

A number of commands can be made to be executed sequentially by explicitly sequencing them as shown below. Such a sequence of commands is itself a command:

```
(command then
 command then
 ... then
 command)
```

7.8.3 *OneOf* Commands and Expressions

One canonical construct for expressing decisions is the `OneOf` command or expression. It chooses code to execute from a number of alternatives, based on the values of a boolean expression associated with each. In the `OneOf` statement, a number of conditional clauses are expressed. Each clause consists of a boolean expression and a body of code. A single `NoneOfAbove` clause may optionally be written after the conditional clauses. It contains a body of code.

```
(OneOf
 (if booleanExpression then body)
 ...
 (if booleanExpression then body)
 (NoneOfAbove body))
```

For example, a OneOf expression for the absolute value of some number `n` (a variable global to the statement) is:

```
(OneOf
 (if (≥ n 0) then n)
 (if (≤ n 0) then -n))
```

Boolean expressions are logical expressions evaluating either to the value `true` or to the value `false`. When a OneOf command is executed, the boolean expressions in all conditional clauses are evaluated concurrently. The body of the first one (temporally) to reply `true` will be chosen for execution. If all clauses have

boolean expressions which return `false`, then the `NoneOfAbove` clause is chosen for execution. If no clause is chosen, and no `NoneOfAbove` clause exists, then the OneOf command is ignored. The same is true for *OneOf* expressions.

OneOf commands must occur in command contexts; OneOf expressions, in expression contexts. All the bodies in a OneOf command must be commands; each body in a OneOf expression must be an expression.

Logical expressions include relational expressions, such as

(> *expression expression*)

and queries to Omega of the form

(*expression* `is` *patternForExpression*)

Because they are expected to be evaluated concurrently, boolean expressions should have only benevolent side-effects. Because Boolean expressions in conditional clauses are evaluated concurrently, it is a good programming practice to ensure that no more than one of the boolean expressions can return true.

7.8.4 *CaseFor* Commands and Expressions

The `CaseFor` command is a construct for expressing decisions based on the result of an expression.

```
(CaseFor expression
         ...
         (is valuePattern then body)
         ...
         (complaint messagePattern then body)
         ...
         (NoneOfAbove body))
```

expression is evaluated, then all of the `is` and `complaint` clauses begin checking (concurrently) whether or not their patterns match the result of the expression. A pattern in an `is` clause matches if *expression* evaluates to some value which is

described by the pattern. A pattern in a **complaint** clause matches if a complaint is generated in the evaluation of *expression*, and the reason in the complaint is described by the pattern. The *body* corresponding to the pattern that is (temporally) first determined to match is chosen, for execution. If no clause has a matching pattern, then the *body* in the **NoneOfAbove** clause is executed.

7.8.5 Simple Conditionals

At times, only simple conditionals are desired. For this, Act-1 allows convenient expression of one-armed and two-armed conditionals. These, too, may be either commands or expressions. They are simply syntactic sugarings of the *OneOf* statement.

In a one-armed conditional, a boolean expression is evaluated. If it returns **true**, the expression or body of commands in its arm will be executed. Otherwise the statement is ignored. A one-armed conditional[12] looks like:

(**if** *booleanExpression* **then** *code*)

A two-armed conditional, a boolean expression (corresponding to the **then** clause) is evaluated as its negation (corresponding to the **else** clause) is[13]:

(**if** *booleanExpression*
  then *code1*
  else *code2*)

---

[12]The one-armed conditional above is equivalent to this Oneof:

(**OneOf**
 (**if** *booleanExpression* **then** *code*))

[13]The two-armed conditional above is equivalent to this OneOf:

(**OneOf**
 (**if** *booleanExpression* **then** *expression1*)
 (**if** (¬ *booleanExpression*) **then** *expression2*))

7.9 Abbreviations for Conventional Communication Handlers

Three kinds of Communications have been discussed. They are the Request, the Reply, and the Complaint. Abbreviations of the general communication handler's syntax exist for conveniently writing handlers for each of these three categories of Communications.

7.9.1 A Request Handler

A handler for Requests is the most involved of the three, because each Request has both a Message and a Customer. A general communication handler for a Request would look like:

```
(is (a Request
    (with message patternForMessage)
    (with customer patternForCustomer))
  communication
  bodyOfCommands)
```

A request handler is a syntactic sugaring of the general communication handler. The pattern in a request handler is matched against the message in the Request. The request's customer is "remembered" by the handler for any responses made within the handler's body. A Request handler has the form

```
(is patternForMessage request bodyCommands)
```

Replying from Within a Request Handler

Replies within the body of the request handler are of the form

```
(Reply message)
```

which sends a Reply with the specified message to the customer "remembered" by the request handler. The `Reply` command above is equivalent to

```
(ReplyTo anonymousCustomer message)
```

Complaining from Within a Request Handler

Similarly, complaints within the body of the request handler have the form

(Complain *message*)

and are equivalent to

(ComplainTo *anonymousCustomer message*)

Written in this more concise notation, the specification for the CheckingAccount example above transliterates to a less verbose version:

```
((a CheckingAccount (with balance (a DollarValue))) is
 (a Behavior
  (is (a BalanceRequisition) request (Reply (a DollarValue)))
  (is (a DepositRequisition (with amount (a DollarValue)))
      request
      (Reply Completed))
  (is (a WithdrawRequisition
            (with amount (a DollarValue))))
      request
      (or (Reply Completed) (Complain Bounced))))
```

7.9.2 Reply and Complaint Handlers

A Reply Handler

A general communication handler for a *Reply* would look like:

(is (a Reply
(with message *patternForMessage*))
communication
bodyOfCommands)

A convenient syntactic sugaring exists for this. We will call this abbreviation a reply handler. Note the fact that it contains the keyword **reply** instead of the keyword **communication**, and the fact that the pattern is one for a message in a reply, instead of one for a communication:

(is *patternForMessage* **reply** *bodyCommands*)

A Complaint Handler

Similarly, a general communication handler for a Complaint would look like:

```
(is (a Complaint
    (with message patternForMessage))
   communication
   bodyOfCommands)
```

Its abbreviation contains the keyword `complaint` rather than `communication`, and the pattern is one for the message within a complaint, rather than that of a communication:

```
(is patternForMessage complaint bodyCommands)
```

For example, consider the familiar actor, `Factorial`. Suppose it accepts a request whose message is an Integer, and that it replies with an appropriate Integer if its argument is greater than or equal to zero; otherwise it complains with a message containing `NegativeArgument`. Consider another actor, `PrintFactorial`, which prints the factorial of an integer sent to it. The implementation of the customer created by `PrintFactorial` is what interests us. `PrintFactorial` will accept a Request whose message is an integer. It will then send a request to `Factorial`. The request will contain the integer in question, as well as a customer, which will accept a response from `Factorial` and will proceed accordingly. Note that this response may either be a Reply or a Complaint, depending on the value of the integer.

```
(PrintFactorial be
 (new Behavior
   (is (a Request
       (with message (=i WhichIs (an Integer)))
       (with customer =c))
      communication
      (SendTo Factorial
        (a Request
         (with message i)
         (with customer
          (a Customer
             (with behavior
              (new Behavior
                (is =w reply PrintWAndReplyToC)
                (is NegativeArgument complaint
                   ComplainToC)))))))))
```

The abbreviated customer, above, looks like this:

```
(a Customer
   (with behavior
    (new Behavior
      (is =w reply PrintWAndReplyToC)
      (is NegativeArgument complaint ComplainToC))))
```

Written in the unabbreviated form, the customer would have looked like this:

```
(a Customer
   (with behavior
    (new Behavior
      (is (a Reply (with message =w))
          communication PrintWAndReplyToC)
      (is (a Complaint (with message NegativeArgument))
          communication ComplainToC))))
```

Chapter Eight

Examples

This chapter illustrates Act-1's language features and their usage. It demonstrates techniques for writing specifications and implementations of actors.

8.1 The Factorial of a Whole Number

Consider the simple factorial operation for Whole Numbers. This operation finds the product of all Whole Numbers from 1 to some specified Whole Number which is greater than or equal to 1. The factorial of zero is defined to be 1.

8.1.1 A Top-Level Specification

A simple characterization of the factorial operation in classical mathematical notation relates the function's domain to its range:

```
Factorial: NonNegativeInteger → NaturalNumber
```

A corresponding Omega description[14] of the attributions of this kind of actor is:

```
((a Factorial (of arg (a NonNegativeInteger)))
          is
          (a NaturalNumber))
```

A description of the behavior of factorials would serve equally well as a specification:

---

[14] In this specification, `arg` is simply a keyword.

```
((a Factorial) is
 (a Behavior
  (is (a Requisition (with arg (a NonNegativeInteger)))
     request
     (Reply (a NaturalNumber)))))
```

In general, the first form is preferred for specifications, when it can conveniently capture enough detail. General relationships between the attribution description form and the communication handler form are evident in this example. A formal set of translations is beyond the scope of this thesis and is the subject of current research. Implementations, too, can be in terms of attributions rather than in terms of communication handling. It is often the case, though, that a communication-handling view is more appropriate or more expressive than an attribute description view (attribute descriptions cannot capture the notion of designating new behavior). Programmers may choose the most convenient forms for specifying and implementing actors. We will proceed in this fashion in the examples to come.

8.1.2 A Simple Recursive Specification of Factorials

The factorial operation is often defined recursively:

```
∀ n ∈ NonNegativeIntegers,
Factorial(n) =
   1                      ,   if n = 0.
   n • Factorial(n - 1),      if n > 0.
```

A corresponding *Omega* specification might consist of two Omega descriptions. The first says that the factorial of zero is 1.

```
((a Factorial (of arg 0)) is 1)
```

The second describes the factorials of all *other* NonNegativeIntegers (the NaturalNumbers).

```
((a Factorial (of arg (=n WhichIs (a NaturalNumber)))) is
 (* n (a Factorial (of arg (- n 1)))))
```

A Self-Contained Specification

This specification can be captured in a single *Omega* description, which contains a decision about whether or not the argument is zero:

```
((a Factorial
    (of arg (=n WhichIs (a NonNegativeInteger))))
 is
 (if (= n 0) then 1
  else (* n (a Factorial (of arg (- n 1))))))
```

8.1.3 Implementations of Factorial

The specification of factorial, above, may have any of a large number of implementations. They can range from a naive one, which looks very much like the self-contained specification above, to an iterative one, to a highly concurrent one, which takes advantage of the concurrent nature of the language.

8.1.3.1 A Simple Recursive Implementation of Factorial

A naive implementation of factorial is a very simple one which looks much like the self-contained specification above. It is simple enough to need no explanation:

```
((new Factorial
     (of arg (=n WhichIs (a NonNegativeInteger))))
 be
 (if (= n 0) then 1
  else (* n (new Factorial (of arg (- n 1))))))
```

A Corresponding Low-Level Implementation

For the reader's satisfaction, we will present a communication-handling alternative to the implementation above. This implementation is for a single actor, `factorial`. This actor can be replicated, because its behavior does not change.

```
(factorial be
 (new Behavior
  ((a Requisition
      (of arg (=n WhichIs (a NonNegativeInteger))))
   request
   (if (= n 0) then (Reply 1)
    else (Reply (* n
             (Ask factorial
                 (a Requisition (of arg (- n 1))))))))))
```

8.1.3.2 An Iterative Implementation of Factorial

Factorial might be implemented iteratively. Iteration in Act-1 is done by tail recursion. An explanation of how it does this can be found in Appendix A.

An IterativeFactorial is simply a loop which accepts a loop index and an accumulator. If the index is zero, the current value of the accumulator is returned. Otherwise, an iteration is performed in which the product of the index and the accumulator replaces the accumulator and the index is decremented. The reader may, at one time, have implemented a similar version of factorial in another language.

```
((new IterativeFactorial
     (of index (=n WhichIs (a NonNegativeInteger)))
     (of accumulation (=a WhichIs (a NaturalNumber))))
 be
 (if (= n 0) then a
  else (new IterativeFactorial
           (of index (- n 1))
           (of accumulation (* n a)))))
```

A Factorial is simply implemented directly by a proper creation of an IterativeFactorial. The appropriate IterativeFactorial is one in which the accumulation is 1 and the index is the argument of the factorial.

```
((new Factorial
      (of arg (=n WhichIs (a NonNegativeInteger))))
 be
 (new IterativeFactorial (of index n) (of accumulation 1)))
```

8.1.3.3 A Concurrent Implementation of Factorial

Factorials might also be implemented in terms of a range product (i.e. the Π function in mathematics). The factorial of a Natural Number is simply the product of the Natural Numbers from 1 to the number. RangeProducts have been defined such that:

```
((a Factorial
    (of arg (=n WhichIs (a NonNegativeInteger))))
 is
 (if (= n 0) then 1
  else (a RangeProduct (of first 1) (of last n))))
```

RangeProducts are interesting in that they decompose the problem of finding the product of a large range of numbers into the problem of multiplying the results of two RangeProduct problems of approximately half the size. Recursively decomposable problems such as this one have much potential for concurrency, because the subproblems can be solved concurrently. Because the behavior of RangeProducts does not change, many copies of a RangeProduct actor can exist and all can process requests concurrently. Here is an implementation for finding the range product from some Natural Number to another Natural Number:

```
((a RangeProduct
    (of first (=lo WhichIs (a NaturalNumber)))
    (of last (=hi WhichIs
                (and (a NaturalNumber) (≥ lo)))))
 is
 (if (= hi lo) then lo
  else
  (let (ave be (/ (+ lo hi) 2)) in
   (* (a RangeProduct (of first lo) (of last ave))
      (a RangeProduct (of first (+ ave 1)) (of last hi))))))
```

8.2 An Addition Operator for Cartesian Complex Numbers

The reader may be wondering about binary operations for abstractions such as ComplexNumbers. One way of handling them follows. A set of common operators such as +, -, and * are overloaded. That is, each operation may have arbitrarily many implementations, each of which can operate on some restricted set of actors, such as Integers or RealNumbers. When Act-1 must interpret one of these operators, it chooses an appropriate implementation for the operator, according to what actors are the arguments of the operator. For example, if the arguments are Integers, the Integer implementation is chosen.

When defining Cartesian Complex Numbers, a programmer might wish to provide a description of each of the standard operators. The definition for the + operator might look like the following, which indicates that two Cartesian Complex Numbers should be added by creating a new Cartesian Complex Number whose real part is the sum of the real parts of the arguments, and whose imaginary part is the sum of the imaginary parts of the arguments:

```
((a +
   (of arg1 (a CartesianComplexNumber
               (with realPart =r1)
               (with imaginaryPart =i1)))
   (of arg2 (a CartesianComplexNumber
               (with realPart =r2)
               (with imaginaryPart =i2))))
 is
 (a CartesianComplexNumber
   (with realPart (+ r1 r2))
   (with imaginaryPart (+ i1 i2))))
```

8.3 A Stack

Stacks are Last-In, First-Out buffers. A high level specification of a Stack might indicate that it may either be an empty stack, or a stack with a `top` which is any actor and a `rest` which must be a stack:

```
((a Stack) same
 (or (an EmptyStack)
     (a Stack
        (with top (an Actor))
        (with rest (a Stack)))))
```

There is a major decision which a programmer might wish to make before proceeding further. That is whether or not a stack should be able to change its behavior. As a quick illustration, consider a stack which has items which have been pushed onto it. Does a "pop" operation merely change the behavior of the stack and reply with the top item, or does it instead reply with the top item and a new stack which results from performance of a pop operation on the original stack. In the latter case, the user would presumably forget about the first stack and remember the new one. Because the behavior of a stack actor does not change in the latter case, it is called *immutable.*

Mutable stacks reduce the burden on the programmer slightly because the programmer does not need to rebind an identifier whenever an immutable stack operation would reply with a new stack. In data structures more complicated than a stack, operations of the immutable structure may have to do some copying. Advantages of the immutable stack is that access to it is not serialized, because its behavior cannot change. For patterns of usage which favor reads, an immutable stack might be more appropriate than a mutable one. We will proceed with implementations of mutable and immutable stacks which support the "push", "pop", and "top" operations.

8.3.1 A Mutable Stack

A Specification of a Mutable Stack

We will specify a stack in terms of the communications it will accept and of the contents of communications it would respond to each. Note the similarity of this specification method to those of specifying parameters and return values of operations in many familiar languages.

A mutable stack might either be an empty stack or a stack with items in it. A stack with items in it replies to a "top" or "pop" with an actor. Such a stack replies to a "push" with a completion report. The fact that such an actor obtains a new behavior as a result of receiving a "push" or "pop" is not reflected in the specification:

```
((a Stack
    (with top (an Actor))
    (with rest (a Stack)))
 is
 (a Behavior
  (is (a TopRequisition) request (Reply (an Actor)))
  (is (a PopRequisition) request (Reply (an Actor)))
  (is (a PushRequisition (with item (an Actor))) request
      (Reply Completed))))
```

An empty stack would accept similar communications, but would respond in a different manner to a "top" or a "pop":

```
((an EmptyStack) is
 (a Behavior
  (is (a TopRequisition) request (Complain Empty))
  (is (a PopRequisition) request (Complain Empty))
  (is (a PushRequisition) request (Reply Completed))))
```

An Implementation of Mutable Stacks

An implementation of an empty stack is not difficult. Any empty stack will complain if sent a "top" or "pop", and will react to a "push" by designating its new behavior to be a new stack whose top is the item from the request and whose rest is an empty stack:

```
((new EmptyStack) be
 (label self
  (new Behavior
   (is (a TopRequisition) request (Complain Empty))
   (is (a PopRequisition) request (Complain Empty))
   (is (a PushRequisition (with item =v)) request
     (become (new Stack (of top v) (of rest self)))
     (Reply Completed)))))
```

An implementation of a non-empty stack is a little more involved, but is not difficult. To a "top" request, it responds with the top item. Upon receiving a "pop" request, it replies with the top item, and designates its new behavior to be the rest of the stack. Upon receiving a "push" request containing an actor to be push-ed, it designates its new behavior to be one whose **top** is the actor from the request, and whose **rest** is the current behavior of the stack. Its implementation is quite straightforward:

```
((new Stack
      (of top =t)
      (of rest (=r WhichIs (a Stack))))
 be
 (label self
  (new Behavior
   (is (a TopRequisition) request (Reply t))
   (is (a PopRequisition) request (Reply t) (become r))
   (is (a PushRequisition (with item =v)) request
      (become (new Stack (with top v) (with rest self)))
      (Reply Completed)))))
```

8.3.2 An Immutable Stack

For every operation in which the mutable stack changed its state, the immutable stack must return a new stack. In that case, the specification for empty stacks would change slightly, in the reaction to a "push":

```
((an EmptyStack) is
 (a Behavior
  (is (a TopRequisition) request (Complain Empty))
  (is (a PopRequisition) request (Complain Empty))
  (is (a PushRequisition) request (Reply (a Stack)))))
```

Similarly, the specification of stacks would indicate that both the "push" and "pop" requests are responded to with a new stack.

```
((a Stack
    (with top (an Actor))
    (with rest (a Stack)))
 is
 (a Behavior
  (is (a TopRequisition) request (Reply (an Actor)))
  (is (a PopRequisition) request
      (Reply (a Pair
                (with item (an Actor))
                (with newStack (a Stack)))))
  (is (a PushRequisition (with item (an Actor))) request
      (Reply (a Stack)))))
```

The implementation of empty stacks is quite predictable, and is very similar to the mutable implementation:

```
((new EmptyStack) be
 (label self
  (new Behavior
   (is (a TopRequisition) request (Complain Empty))
   (is (a PopRequisition) request (Complain Empty))
   (is (a PushRequisition (with item =v)) request
     (Reply (new Stack (of top v) (of rest self)))))))
```

The immutable implementation of stacks also resembles the implementation of mutable stacks. Notice that instead of designating a new behavior to replace its current behavior, it includes the new behavior in a reply.

```
((new Stack
      (of top =t)
      (of rest (=r WhichIs (a Stack))))
 be
 (label self
  (new Behavior
   (is (a TopRequisition) request (Reply t))
   (is (a PopRequisition) request
       (Reply (a Pair
                 (with top t)
                 (with newStack r))))
   (is (a PushRequisition (with arg =a)) request
       (Reply (new Stack
                   (with top a)
                   (with rest self)))))))
```

8.4 A Mutable Symbol Table

A more interesting data structure is a symbol table. We will only demonstrate a mutable implementation. The reader should be able to envision or implement an immutable version without major problems.

Specification of a Mutable Symbol Table

At top level, our symbol tables look like:

```
((a SymbolTable) same
 (or (an EmptySymbolTable)
     (a SymbolTable
        (with symbol (an Actor))
        (with value (an Actor))
        (with rest (a SymbolTable)))))
```

At the communication-handling level, our symbol tables will respond to "get", "assign", and "purge" requisitions. "get" will request the value associated with a symbol. "purge" will request that a symbol be removed from the symbol table. "assign" will request that the symbol table add a new symbol-value binding to itself. A specification of a symbol table looks like:

```
((a SymbolTable
    (with symbol =s)
    (with value =v)
    (with rest =r))
 is
 (a Behavior
  (is (a GetRequisition (with symbol =s1)) request
      (or (Reply (an Actor)) (Complain NotFound)))
  (is (a PurgeRequisition (with symbol =s1)) request
      (or (Reply Completed) (Complain NotFound)))
  (is (an AssignRequisition
       (with symbol =s1)
       (with value =s2))
     request
     (Reply Completed))))
```

An empty symbol table would respond to either of the "get" and "purge" requisitions with a complaint, and would reply to an "assign" requisition with a completion report:

```
((an EmptySymbolTable) is
 (a Behavior
  (is (a GetRequisition) request (Complain NotFound))
  (is (a PurgeRequisition) request (Complain NotFound))
  (is (an AssignRequisition
       (with symbol =s1) (with value =v1))
      request (Reply Completed))))
```

An Implementation for Symbol Tables

One can envision many and varied implementations for symbol tables as specified above. Implementation strategies could include linked lists, B-trees, binary trees, and hash tables.

For the sake of simplicity and convenience, we will present a linked-list flavor of implementation. Here is the implementation strategy. Envision a symbol table as an actor with a symbol, a value, and another symbol table. If that actor receives a request, it can check if the symbol in the request is the same as its symbol. If so, it can handle the request; otherwise, it can just pass the request on to the next symbol table. The empty symbol table at the end of the chain handles any requests which reach it.

If an empty symbol table receives either a "get" or a "purge" requisition, it will complain. It it receives an "assign" request with a symbol and a value, it will designate a replacement behavior to be a new symbol table with the given symbol and value and with the empty symbol table as its next symbol table. Here is the implementation in Act-1:

```
((new EmptySymbolTable) be
 (label self
  (new Behavior
   (is (a GetRequisition) request (Complain NotFound))
   (is (a PurgeRequisition) request (Complain NotFound))
   (is (an AssignRequisition
         (with symbol =s) (with value =v))
      request
      (become
       (new SymbolTable
         (with symbol s)
         (with value v)
         (with rest self)))))))
```

The implementation of a non-empty symbol table becomes very easy. A symbol table actor satisfies a request if its symbol is the one in question; otherwise, it passes the request on to the next symbol table.

```
((new SymbolTable
   (with symbol =s)
   (with value =v)
   (with rest (=r WhichIs (a SymbolTable))))
 be
 (new Behavior
  (is (=r1 WhichIs
    (a Request (with customer =c)
       (with message (a GetRequisition (with symbol =s1)))))
      communication
      (if (= s s1) then (ReplyTo c v)
       else (SendTo r r1)))
  (is (=r1 WhichIs
    (a Request (with customer =c)
       (with message
          (a PurgeRequisition (with symbol =s1)))))
      communication
      (if (= s s1) then (become r) (Reply Completed)
       else (SendTo r r1)))
  (is (=r1 WhichIs
    ((a Request (with customer =c)
        (with message
           (an AssignRequisition
               (with symbol =s1)
               (with value =v1)))
      request
      (if (= s s1) then
         (become (new SymbolTable
                     (with symbol s)
                     (with value v1)
                     (with rest r)))
         (Reply Completed)
       else (SendTo r r1)))))
```

Chapter Nine

Actor Programming Methodology

9.1 Guardians for Protection of Shared Resources

The Shared Resource Problem

The problem of managing shared resources is an important one in Computer Science. Shared resources may be hardware, such as a printer, a secondary storage device, or a communications network. Shared resources may be portions of a memory space, such as memory space representing a dictionary object or a bank account. Of particular interest is ensuring correct access to the resources even in the face of concurrent access by independent actors.

The Client-Based Synchronization Scheme

One possible resource management scheme is to let the user of a resource gain control of it, use it, then yield control. This requires that each user know the details of how the resource must be accessed, and how control of the resource is attained and yielded. If the resource is changed (for example if the printer is replaced with a new model or the representation of an object is changed), all users of the resource must be made aware of the change. Correct use of the resource can only be ensured by verifying all the users of the resource.

The Guardian Scheme

Another resource management scheme allows only a single module, called the *guardian* of the resource, to access the resource. All information related to the details of accessing the resource are encapsulated within the guardian. Users of the resource communicate their demands to the guardian, who processes the requests,

then communicates the results back to the users. The processing of a request may result in refusal, negotiation, or accessing of the resource.

Encapsulation of Detail and Protocol

The guardian scheme has important advantages over the first. In the guardian scheme, correct access to the resource depends only on the composition of the guardian. The guardian is in charge of synchronizing access to the shared resource, so all details of the access and of the synchronization can be hidden within the guardian. It has been found that depending upon a user to use conventional protocols is more error-prone than encapsulating the details of protocol behind a simple interface.

Encapsulation of details of access to a shared resource makes verification of the correctness of a system of actors which make use of the resource more tractable. Rather than to ensure that each and every user correctly access the resource, programmers need only verify that the guardian has the correct behavior.

Encapsulation of access detail makes adjustment to change easier. Changes in conventions for accessing shared resources need not spread beyond the implementation of the resource's guardian. In the first model, each user or client would have to be changed.

Access to the resource is arbitrated by the communication-handling mechanisms of the guardian. No additional synchronization mechanisms need to be programmed.

Guardians are an Abstraction Mechanism

An actor can be viewed as a guardian for a shared resource which is a behavior. A guardian may be constructed out of a system of actors, coordinated to perform a resource-protecting function. A guardian system looks like an actor to a

user of the system. Guardian systems, then, can be constructed out of actors and other guardian systems.

9.2 Thoughts on an Actor Design Methodology

Design of actor-based software systems is similar to design using any other language which supports high-level abstractions [Hewitt 76]. Good design methodologies promote the use of abstractions natural for the function performed, making the software system appear to be a hierarchy of modules forming an abstract machine which is geared to perform those functions.

The design process is one of creating a hierarchy of abstractions, then walking through the hierarchy, refining the specifications of the abstractions or modifying the hierarchy's structure as appropriate. The initial phase requires three basic types of activity. Decide what kinds of actors are natural for the system to be constructed. For each actor, decide what kind of communications it should accept. For each actor, decide what it should do when it receives each kind of acceptable communication.

These design decisions describe the ***behavior*** of the system. The data structures and control structures of the implementation should be determined by these design decisions, and not by arbitrary language limitations.

9.2.1 Composition of Guardians

Guardians which perform complex functions may be composed of a system of actors, instead of a single actor. Five types of actors have been found useful in the creation of composite guardians. They are listed below.

~ **The Guardian** -- presents the user interface of the guardian system. It

initially constructs the other modules, as well as a suitable environment for them.

~ **Servers** -- Each server performs whatever manipulations are necessary to enqueue requests from customers, or to relay completion reports back to the customers. After the Server handles any communication, it invokes a Transition, then becomes the server constructed by the Transition to be its replacement.

~ **Transitions** -- Each Transition ponders the implications of any state change caused by the handling of a communication by a Server. It performs whatever operations are necessary to convert the state of the server to a desirable one, then constructs a replacement server for the server, which it sends in its reply. For example, if a resource becomes idle, and there are pending requests pending for the resource, the Transition dequeues a request, ships the request off to the resource, and marks the resource as busy.

~ **Schedulers** -- A Server or Transition may invoke a Scheduler to order the handling of communications, once they have been accepted. Such ordering may occur as a Server enqueues requests, for example, or as a Transition dequeues a pending request.

~ **TransactionManagers** -- Each transaction manager handles whatever transactions are necessary with the customer and with the resource in order to allow a request for the resource to be handled for the customer. For example, the TransactionManager might authenticate the customer, or request funds for use of the resource. The TransactionManager reacts to requests and responses from the resource, such as CompletionReports (which are relayed to the Server), or special types of Complaints (such as OutOfPaper reports from a printer).

Appendix A

Iteration as Tail Recursion

Here is an iterative version of the factorial program, with iteration via tail-recursion. The following sets up the iterative portion of the algorithm:

```
(factorial be
 (new Behavior
    (is (=n WhichIs (a WholeNumber))
     Request
     (Ask iterativeFactorial
          (a Requisition
             (with index n)
             (with accumulation 1))))))
```

and the following is the iterative portion:

```
(iterativeFactorial be
 (new Behavior
    (is (a Requisition
        (with index (=i WhichIs (a WholeNumber)))
        (with accumulation (=a WhichIs (a WholeNumber))))
     Request
     (CaseFor i
      (is 0 then (Reply a))
      (is (> 0) then
         (Reply
          (Ask iterativeFactorial
               (a Requisition
                  (with index (- i 1))
                  (with accumulation (* i a))))))))))
```

Here is a translation of `iterativeFactorial` which does not elide the customer.

```
(iterativeFactorial be
 (new Behavior
  (is (a Request
      (with message
        (a Requisition
         (with index (=i WhichIs (a WholeNumber)))
         (with accumulation (=a WhichIs (a WholeNumber)))))
      (with customer =c))
   Communication
   (CaseFor i
    (is 0 then (ReplyTo c a))
    (is (> 0) then
      (SendTo iterativeFactorial
        (a Request
           (with message
             (a Requisition
               (with index (- i 1))
               (with accumulation (* i a))))
           (with customer
              (a Customer
                 (with behavior
                  (new Behavior
                   (is =r reply (ReplyTo c r))))))))))))
```

Note that the customer merely relays the answer to c. It is worthwhile simply having the customer of the request be c, itself, rather than the relay.

```
(iterativeFactorial be
 (new Behavior
  (is (a Request
      (with message
        (a Requisition
         (with index (=i WhichIs (a WholeNumber)))
         (with accumulation (=a WhichIs (a WholeNumber)))))
      (with customer =c))
   Communication
   (CaseFor i
    (is 0 then (ReplyTo c a))
    (is (> 0) then
      (SendTo iterativeFactorial
        (a Request
           (with message
             (a Requisition
               (with index (- i 1))
               (with accumulation (* i a))))
           (with customer c)))))))))
```

Eliminating the relay means that no new actors are created as

IterativeFactorial computes. Therefore, we find that **iterativeFactorial** is, in fact, iterative.

In general, a nested expression of the form

(ReplyTo c (Ask *actor message*)**)**

is transformed into an expression of the form

```
(SendTo actor
   (a Request
      (with message message)
      (with customer c)))
```

This saves the overhead of creating a customer when translating the **Ask** which simply relays the response to the actor **c.**

Bibliography

[Attardi and Simi 81]
Attardi, G., and M. Simi.
Consistency and Completeness of a Logic for Knowledge Representation.

[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):613-641, August, 1978.

[Barber 82]
Barber, G.R.
Office Semantics.
PhD thesis, Massachusetts Institute of Technology, 1982.

[Brooks 75]
Brooks, F.P.
The Mythical Man-Month: Essays on Software Engineering.
Addison-Wesley, 1975.

[Clinger 81]
Clinger, W.D.
Foundations of Actor Semantics.
PhD thesis, Massachusetts Institute of Technology, 1981.

[Fahlman 79]
Fahlman, S.E.
NETL: A System for Representing and Using Real-World Knowledge.
MIT Press, 1979.

[Hewitt and Attardi 81]
Hewitt, C.E. and G. Attardi.
Guardians for Concurrent Systems.

[Hewitt and Baker 78]
Hewitt, C.E., and H. Baker.
Actors and Continuous Functionals.
In Neuhold, editor, *Formal Description of Programming Concepts.* North Holland, 1978.
Also available as MIT/LCS/TR-194

[Hewitt, Attardi and Lieberman 78]
Hewitt, C.E., G. Attardi, and H. Lieberman.
Delegation in Message Passing.
In *First International Conference of Distributed Systems.* Huntsville, 1978.

[Hewitt, Attardi and Simi 80]
Hewitt, C.E., G. Attardi, and M. Simi.
Knowledge Embedding in the Description System Omega.
In *Proceedings of the AAAI.* AAAI, August, 1980.

[Hewitt 76]
Hewitt, C.E.
Viewing Control Structures as Patterns of Passing Messages.
AI Memo 410, Massachusetts Institute of Technology, December, 1976.

[Hewitt 80]
Hewitt, C.H.
The Apiary Network Architecture for Knowledgeable Systems.
In *Lisp Conference.* Stanford, August, 1980.

[Hoare 78]
Hoare, C.A.R.
Communicating Sequential Processes.
Communications of the ACM 21(8):666-677, August, 1978.

[Horsley and Lynch 79]
Horsley, T.R. and W.C. Lynch.
Pilot: A Software Engineering Case Study.
7th Symposium on Operating Systems Principles, December, 1979.

[Kerns 80]
Kerns, B.S.
Towards a Better Definition of Transactions.
Available as MIT AI Memo number 609.

[Kleinrock 76]
Kleinrock, L.
On Communications and Networks.
IEEE Transactions on Computers

[Kornfeld and Hewitt 81]
Kornfeld, W.A. and C.E. Hewitt.
The Scientific Community Metaphor.
IEEE

[Lieberman 81a]
Lieberman, H.
A Preview of Act-1.
Available as MIT AI Memo number 625.

[Lieberman 81b]
Lieberman, H.
Thinking About Lots Of Things At Once Without Getting Confused:
Parallelism in Act-1.
Available as MIT AI Memo number 626.

[Liskov, et al 79]
Liskov, B.H., et al.
Clu Reference Manual.
Technical Report 225, Massachusetts Institute of Technology, October, 1979.

[Nygaard and Dahl 66]
Nygaard, K. and O. Dahl.
Simula: An Algol-Based Simulation Language.
Communications of the ACM 9(9), September, 1966.

[Pouzin and Zimmerman 78]
Pouzin, L. and H. Zimmerman.
A Tutorial on Protocols.
Proceedings of the IEEE

[Wulf, London and Shaw 76]
Wulf, W.A., R.L. London, and M. Shaw.
Abstraction and Verification in Alphard: Introduction to Language and
Methodology.