Technical Report 728

Issues in the Design and Implementation of Act2

Daniel G. Theriault

Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

Issues in the Design and Implementation of Act2

by

Daniel Gary Theriault

Massachusetts Institute of Technology

June 1983

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the research reported on in this paper was provided primarily by the System Development Foundation. Support for the Artificial Intelligence Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N0014-80-C-0505.

This empty page was substituted for a blank page in the original document.

Issues in the Design and Implementation of Act2

by

Daniel Gary Theriault

Revised version of a thesis submitted to the
Department of Electrical Engineering and Computer Science
on May 6, 1983 in partial fulfillment of the requirements
for the Degree of Master of Science

Supervised by Professor Carl E. Hewitt

Abstract

Act2 is a highly concurrent programming language designed to exploit the processing power available from parallel computer architectures. The language supports advanced concepts in software engineering, providing high-level constructs suitable for implementing artificially-intelligent applications. Act2 is based on the Actor model of computation, consisting of virtual computational agents which communicate by message-passing. Act2 serves as a framework in which to integrate an actor language, a description and reasoning system, and a problem-solving and resource management system. This document describes issues in Act2's design and the implementation of an interpreter for the language.

This empty page was substituted for a blank page in the original document.

Acknowledgments

I would like to take this opportunity to shower my thanks on Carl Hewitt, my thesis advisor, for his help in making this thesis a reality. He provided a wealth of ideas and encouragement. My wife, Candace, provided moral support and helped massage drafts into more coherent forms, for which I will be forever grateful. I hope I will be able to make up for this period of intense suffering during the years ahead. I'd like to extend special thanks to Henry Lieberman and Jonathan Amsterdam for their implementation of Scripter and an Apiary, without which an Act2 interpreter would remain a dream. Peter de Jong, Carl Mikkelsen, Gene Ciccarelli, Dan Weld, Roy Nordblom, and Priscilla Cobb also played essential roles in forming that dynamic (and unique) environment called the Message Passing Semantics group. This thesis would have been impossible without the solid foundation of Carl Hewitt's work, Henry Lieberman's implementation of Act1, Jeff Schiller's work on Apiary0, Bill Kornfeld's Ether, and work on Omega by Jerry Barber, Beppe Attardi, and Maria Simi. I owe a debt of gratitude to Charles Smith and the System Development Foundation for their financial support.

This empty page was substituted for a blank page in the original document.

To my wonderful wife, [illegible]

This empty page was substituted for a blank page in the original document.

Preface: A Guide to this Document

The organization of this document was part of an attempt to satisfy a variety of audiences. When possible, essential information is encapsulated in a convenient location, to be studied or ignored as a whole. The main body of the document describes the historical setting in which Act2 materialized, the creation process, and a rationale for its design. The appendices generally describe the language itself, and serve as reference material.

Chapter One describes the context in which Act2 was built, including the foundation of previous work upon which it stands. Chapter Two is an impressionistic introduction to the language itself, making use use of canonical examples. At this point, interested readers may browse through the appendices in order to become more familiar with the language before pushing on. The following chapters assume a familiarity with the language and a willingness to refer to appendices for details of the language's syntax and semantics. Chapter Three relates the design and development strategies used to produce Act2. Chapter Four discusses issues considered important in Act2's design. It is the backbone of the document's body. Chapter Five touches on implementation issues and mechanisms, and Chapter Six wraps up with a summary and conclusions.

Appendix A contains a glossary for help in decoding actor jargon. Appendix B presents a sample of conversational interaction with Act2. Appendix C informally describes the syntax and semantics of Act2, construct by construct. Appendix D is a more formal description of the language, in the form of an Act2 implementation of itself. It is useful for resolving ambiguities in the natural language descriptions of Act2, and for understanding the general strategies used in its implementation.

Appendix E describes predefined names and actors which come with an installation of Act2, as well as some standard communication protocols which they use. Appendix F discusses a few more language issues considered too distracting or unimportant for Chapter Four.

Table of Contents

Chapter One

Conceptual Framework

The recent history of Computer Science shows significant advances in computer software and hardware engineering. Increasingly sophisticated and complex software application systems are being designed and implemented, especially in the area of Artificial Intelligence. Requirements for software have grown to include open systems, in which autonomously owned and independently conceived software systems communicate and cooperate. Modern programming languages may exploit the increased parallelism afforded by hardware and support the software engineering principles and practices for reduction of complexity in designing and implementing software systems.

These trends were anticipated by [Hewitt 77, Hewitt and Smith 75], which proposed a novel computational model, based on virtual computational agents called actors. The actor model was abstraction-oriented, processor-independent, and inherently concurrent. Languages realizing this model are intended to exploit parallelism available in future computer architectures.

The first actor language, Plasma, was essentially an experiment to determine whether it was possible to construct a language based on the actor model of computation. Though Plasma was a useful language in itself, its design and implementation pointed out the fact that more needed to be learned about actor-based languages with advanced features suitable for Artificial Intelligence applications. It also pointed out that trying to solve the whole problem at once was not a practical approach; that it may be more wieldy to decouple some of the issues

and mechanisms by experimenting with different aspects of the problem more independently.

The Act1 programming language [Lieberman 81a] was a direct realization of the actor computational model. It was an experiment in the use of actors and in expressing their behavior and communication among them. The Omega description and deduction system [Barber 82, Hewitt, Attardi, Simi 80] was an experiment in knowledge representation and manipulation mechanisms useful for languages implementing artificial intelligence applications. Ether [Kornfeld 79] was a reasoning system for solving problems in much the same way they are solved by scientific communities. It dealt with the creation and management of independent problem solvers cooperating to establish or refute common goals. Apiary0 [Hewitt 80] was a design for a computer architecture consisting of a large number of independent processors interconnected with high-bandwidth links. The computer architecture itself was responsible for services such as storage management, transmission of communications, migration of actors, and load-balancing. Languages built of top of an Apiary can ignore such issues.

Many new ideas and insights were acquired in the design and development of each of these experimental systems. Now that they have been completed, the time has come to integrate these ideas and others developed independently into a single, more sophisticated programming language: Prelude.

The Act2 programming language is the first step in implementing Prelude. It blends basic ideas, mechanisms, and philosophies from Act1, Omega, and Ether in a single programming language. They are not simply juxtaposed, but permeate the language through to its foundation. Act2 itself does not fully implement the more sophisticated aspects of Omega and Ether, but is extensible in a manner such that the rest of Prelude can be embedded within it.

Our implementation of Act2 runs on Lisp Machines [Weinreb and Moon 81]. It is written in Scripter [Lieberman 83], a language embedded in Lisp Machine Lisp, tailored for expressing actor computations.

1.1 The Actor Model of Computation

Early computational models were significantly more machine-oriented than the actor model. Early languages implicitly had a model in which computation progressed as a succession of modifications to a global machine state. Both the existence of a set of fixed-size storage locations and a set of machine instructions showed through to the language level. Data structures were mapped onto sets of contiguous storage locations. Procedures were developed to encapsulate a series of primitive operations, procedure calls, and state changes as a single abstract operation. Object-oriented languages abstracted away the structure of the store. An object consisted of some storage and primitive operations with which to access and manipulate this concrete representation. Though encapsulating the representation of data types was a tremendous advancement, the underlying computational paradigm was still that of sequentially modifying a global state. Advancements in hardware technology have provided increasing amounts of parallelism for programming languages to exploit. Languages based on the old computational models are inherently sequential, and need special attention to exploit parallelism.

1.1.1 Actors

The actor model of computation [Hewitt and Baker 78] is one in which many active, self-contained computing entities, called *actors*, process communications in parallel. Each actor has its own processing power and storage. Instead of having a notion of control flow, the actor model makes use of a more flexible idea of

cooperation; of communication among entities which are under their own control. Actors interact by transmitting information in *communications* to each other.

An actor is a mathematical abstraction [Clinger 81a]. It is self-contained and opaque in the sense that its internal composition cannot be directly seen or manipulated by other actors. They are restricted to sending communications to the actor and observing whatever communications the actor might send in reply. Only the actor itself can access its underlying representation. It also is responsible for how it reacts to any communication; it may even choose to request authentication, request additional computing resources, or reject the communication altogether. An actor is an encapsulation mechanism providing information-hiding capabilities, which are a corner stone of good software engineering.

Each actor has a *script*, which determines what communications it can accept and what computations it will perform upon receiving each. It may also have some *acquaintances*, which are other actors it can directly communicate with as it processes a communication. An actor's *behavior* is uniquely characterized by its script and acquaintances. When it accepts a communication, an actor can make simple decisions, create new actors, send communications to its acquaintances (or to itself), and designate an actor to serve as a replacement for itself.

One of the effects an actor can cause is the replacement of itself by another actor. It becomes indistinguishable from the replacement actor, which processes any future communications for the actor. *Serialized actors*, or *serializers*, are actors which may change. *Unserialized actors* are actors whose behavior includes no provision for change. The distinction is a very important one. Because a serializer may change as a result of processing a communication, it can only process one communication at a time. For this reason, the order of arrival of communications is important for serializers.

Unserialized actors, on the other hand, can change neither their behavior nor their acquaintances, and as a result can process communications concurrently. Arrival ordering does not matter, because behavior does not change. Unserialized actors can also be copied arbitrarily, because lack of change will make the copies indistinguishable.

1.1.2 Transactions

Communications are also actors. There are three kinds of communications, representing the major forms of communication in transactions among actors. Each communication has a *message* acquaintance containing information for the target actor. An actor can send a *request* communication to another, asking it to cause effects or provide information of some form. After the request has been successfully fulfilled, some actor will eventually respond to the request with a *reply* communication. Otherwise, the response is a *complaint* communication containing a message, which says why the request could not be successfully processed. We refer to replies and complaints collectively as *responses.*

Transmission of communications is one-way, asynchronous, and buffered. Concurrent activities can be spawned simply by transmitting more than one communication when processing a communication. The sender does not wait for the receiver (or *target*) to be ready to receive a communication; instead, the communication is enqueued for reception by the receiver. If the receiver is serialized, arrival order is preserved in a first-in, first-out queue. The sender of a request does not wait for a response from the receiver, because all communication is one-way. Instead, the sender includes in the request a *customer*, an actor to which a reply can be sent. It also includes a *complaint department*, to which a complaint can be sent, in the event that the request cannot be satisfied. When an actor sends a request to another actor, it includes in the request a customer and complaint

department, which are responsible for completing the computation. While this computation continues, the actor might begin processing another communication. See [Kerns 80] for a rigorous definition of transactions.

Computation is event-driven. An *event* happens when an actor accepts a communication for processing. An actor only consumes computing resources when it processes a communication. An event is machine-independent, because all of the information necessary to process it is present in the incoming communication and target actor — its behavior and acquaintances. A transaction begins by sending a request to some actor, which might send communications to other actors. Eventually, an actor might reply to the original customer or complain to the original complaint department.

Because of its emphasis on communication, the actor model of computation unifies the ideas of procedural, data, and control abstraction developed by languages using other models. For example, a data abstraction, such as a checking-account, can be embodied in an actor with an acquaintance that serves as a current balance and with a behavior that responds appropriately to requests for deposits, withdrawals, and balances. A procedural abstraction, such as factorial, can be embodied in an unserialized actor which accepts a request containing an integer, performs a computation (possibly asking itself for the factorial of other integers), then replies with the result. Control abstractions such as recursion, iteration, back-tracking, tree traversal, etc. can be embodied in actors which send each other appropriate communications.

1.2 Plasma

Plasma was the first actor language. As the first language design endeavor using the actor model of computation, it made some progress in implementing and developing the model. At that time, the actor model was in its infancy, and advancements have since been made, thanks to experiments such as Plasma and Act1.

Plasma had basic facilities for transmitting communications, but did not formally distinguish requests, replies, and complaints as different kinds of communications. It incorporated the ideas of expressing control structures as patterns of message-passing, and of unifying the notions of data structures and procedures by concentrating on communication. The language had simple data structures such as numbers and simple constructors like sequences and packagers. Packagers were similar to record structures in languages such as Pascal, allowing the encapsulation of a set of labeled actors, but lacked the flexibility and power of instance descriptions developed in Omega. Though Plasma did acknowledge the need for change, the idea of serializers had not yet been conceived and formalized.

1.3 Act1

Act1 was a programming language which directly realized the actor model of computation [Lieberman 81a, Lieberman 81b]. It was implemented in Maclisp for PDP10, as an experiment in implementing an actor language which uses the message-passing paradigm down to the level of primitive actors, such as numbers and lists. It helped formalize common patterns of message-passing and useful types of communications, as well as the notion of change in actors. As an experimental language, it was unencumbered with mechanisms such as those in Omega and Ether, which provide sophisticated services for the programmer. It provided mechanisms

for creation of actors and for point-to-point communication between actors. It allowed an actor to delegate its incoming communications to another actor for handling.

Act1 allowed a programmer to write programs which appeared to have two-way communication between actors, and translated such expressions into requests with appropriate customers and complaint departments.

Act1 provided constructs for sending arbitrary communications to actors. It also provided constructs for actors to change their behavior, and provided explicit synchronization primitives to avoid problems of change. Act1 also provided a notion of a guardian, an actor which could accept requests, store away state information, then reply to their customers at some later time.

Sub-expressions in Act1 were evaluated sequentially. However, Act1 provided the following constructs for lazy and eager evaluation of expressions:

(delay *expression*)
(hurry *expression*)

When evaluated, the **hurry** expression would create and reply with a future, which was an actor representing the value of the expression inside the **hurry** expression. A newly-spawned process would evaluate that expression concurrently with whatever activity occurred once the future was returned. If the future ever became inaccessible, the process computing the expression's value could be garbage-collected. If any communications were sent to the future actor, it would enqueue them, then send them to the result of the expression, once its evaluation terminated.

In addition, Act1 had the notion of a race for concurrent activity. Given a list of expressions to be evaluated, a result list was immediately provided. As results became available, they were appended to the list asynchronously. An actor with

such a race in its possession could apply the standard first and rest operations on it. Synchronization was done by the race, so that if results were not yet available, it would wait for them before responding. If the race became inaccessible, it along with the processes still computing for it would be garbage-collected.

Act1 had primitive actors, such as numbers and symbols. It also had constructors, like sequences. It had a form of constructor, called a package, which resembled Plasma packagers and behaved in essentially the same manner. Pattern-matching was performed as a structural correspondence between the pattern and object of the match.

1.4 Omega

Omega [Hewitt, Attardi, Simi 80, Attardi, Simi 81, Barber 82]is a system for representing knowledge in general, reasoning about knowledge, and retrieving information from a knowledge base. It represents knowledge as *descriptions* representing abstract concepts and individuals, and as relationships among those descriptions.

The simplest form of Omega description is an *atomic description*, such as **real-number**, **complex-number**, **12**, **man**, **animal**, or **Jack**. These represent abstract concepts or individuals in a model of some world.

An *instance description* represents some collection of individuals which are instances of some abstract *concept*. It can also be thought of as representing any individual in such a collection. Examples of simple instance descriptions include:

```
(a real-number)
(a complex-number)
(a man)
(an animal)
```

The collection of individuals represented by an instance description can be restricted by describing *attributes* which they possess. The order in which attributes appear in an instance description is irrelevant.

```
(a man (with mother Jill))
(a man (with mother (a woman)) (with father (a man)))
(a man (with mother (a woman (with father Bill))))
```

One fundamental mechanism for knowledge representation is the assertion of an *inheritance* relationship between two descriptions. This represents a relationship between the collections of individuals the descriptions represent. The statement `(D1 is D2)` represents the idea that anything described by the description `D1` is also described by the description `D2`. This also means that `D1` is a *specialization* of `D2`; that `D2` is a *generalization* of `D1`. Any knowledge associated with `D2` is inherited by `D1`.

```
(Jack is (a man))
(man is (a species))
((a man) is (a mammal))
(12 is (a real-number))
((a real-number) is (a complex-number))
((a real-number) is (a complex-number (with imaginary-part 0)))
```

Omega implements an omega order logic for making deductions about generalizations and specializations. Omega uses several axioms for reasoning about descriptions and the inheritance relationships between them. For example, there is an axiom establishing the commutativity of attributes, so that it doesn't matter in what order they are included in the instance description. There is an axiom establishing the transitivity of the inheritance relation. There are axioms for dealing with conjunctions and disjunctions of descriptions, and for relating them to single descriptions. In addition to the *with* attributes above, there are different kinds of attributes, to which more or fewer axioms can be applied.

For example, the relationship `(Jack is (a mammal))` can be deduced from the relationships `(Jack is (a man))` and `((a man) is (a mammal))` by the

transitivity axiom. Very significant amounts of knowledge can be embedded in a lattice formed by descriptions and inheritance relationships between them.

A description lattice is required to be monotonic. All descriptions are unchanging, all inheritance relationships are assumed to hold forever once asserted, and knowledge can be added but never altered or removed. Revision of beliefs, opposing sets of beliefs, and suppositions can be represented using a viewpoint mechanism. Omega assertions are commutative — the same deductions can be made, no matter what order the inheritance relationships are asserted.

Omega supports partial description of abstract ideas, and incremental specialization of those descriptions with others. Knowledge about the characteristics of an abstract concept or object and its relationship to other concepts can be embedded in an incomplete fashion, and new pieces of knowledge can be added incrementally. Later, we can retrieve and use not only the particular information asserted, but combinations thereof and deductions made from them using sets of assertions made in the knowledge base and sets of axioms for relating them to each other.

1.5 Ether

Ether is a highly concurrent problem solving system based on a metaphor of problem-solving in a scientific community [Kornfeld 79, Kornfeld, Hewitt 81, Kornfeld 82]. In this model, many independent problem solvers (called *sprites*) can exist in a community. Each sprite specifies a computation to be performed if some specific goal or hypothesis is presented to the community for comment. Sprites can either work to prove or disprove a *goal.* Such a computation can post new goals or *hypotheses.* One of the fundamental concepts is *dissemination* of

information to all sprites which have an interest in it.

Ether has a resource management scheme which associates a *sponsor* with each goal. The sponsor provides computing resources for those working to prove or disprove the goal. Sprite must request some of these resources, which are used as its computation proceeds.

Ether contains commands for disseminating goals and hypotheses. For example, assuming we had primitives for description and inheritance like those in Omega, we might represent the fact that some pattern **pattern0** matched some object **object0** with a predicate: **(pattern-matches pattern0 object0)**.

We might disseminate this as a goal:

```
(disseminate (goal (pattern-matches pattern0 object0))).
```

Some sprite we create then activate may eventually prove this, then disseminate it as a hypothesis:

```
(disseminate (hypothesis (pattern-matches pattern0 object0))).
```

Alternatively, some sprite may disprove it, in which case it may disseminate its negation:

```
  (disseminate (hypothesis (not (pattern-matches pattern0 object0)))).
```

We can create and activate a trivial sprite with an expression such as:

```
(disseminate
  (when (goal (pattern-matches =p =o))
    (if (eq p o)
      then (disseminate (hypothesis (pattern-matches p o))))))
```

Other sprites might try to prove or disprove this goal in other ways. For example, if the pattern and object are instance descriptions, one sprite might try to establish a simple correspondence between the concepts and attributes. Another

might traverse an Omega lattice, trying to find a more elaborate way of matching the pattern and object. Other sprites might be responsible for applying Omega axioms to descriptions and inheritance relations. Note that Ether itself knows nothing about instance descriptions, Omega lattices, inheritance relationships, or deduction axioms. These are assumed to be accessible via some operations independent of Ether itself.

Often, a sprite trying to establish a goal will disseminate sub-goals, then disseminate a hypothesis once the sub-goals have been established. Large numbers of hypotheses may be disseminated as a result. Because these have been disseminated, they can be used in establishing other goals, without being re-established. Because Ether is monotonic, these hypotheses are never forgotten. An interesting phenomenon results, called combinational implosion, like the gains obtained from dynamic programming techniques.

Ether allows programmers to explicitly mention sponsors and viewpoints. For example, a goal can be sponsored by some specified sponsor or with respect to some specified viewpoint:

```
(disseminate
  (goal (pattern-matches pattern0 object0)
        (with sponsor sponsor0)
        (with viewpoint viewpoint0))
(disseminate
  (when (goal (pattern-matches =p =o)
              (with sponsor =s)
              (with viewpoint =v))
    ...))
```

Constructs exist for establishment of sophisticated resource management policies and for establishing relationships among viewpoints. One command for resource management simply makes a sponsor refuse to provide resources to sprites requesting them. This is useful for staving off alternative proofs of a goal once the goal has been established or disproved. For example, if the goal above had really

been disseminated, and a sprite had just established it, the sprite could choke off related computation by other sprites with the command:

```
(withhold sponsor0
  (with reason (established (pattern-matches pattern0 object0))))
```

Ether supports pluralism. Conflicting hypotheses can exist freely in different viewpoints. Sprites can try to establish the same goal with different approaches. Some sprites can try to establish a goal while other sprites can try to refute it. Ether relies on monotonicity. It assumes that any hypotheses once disseminated will remain available and unchanged for all time thereafter. Ether supports commutativity. Goals and hypotheses disseminated after the activation of a sprite will be made available to the sprite and processed as appropriate. Likewise, sprites activated after the dissemination of goals and hypotheses will be available for the sprite. Ether has much potential for parallelism. The very notion of sprites is as problem solvers which compute independently, and therefore concurrently. The monotonicity criterion avoids synchronization problems. The unit of concurrency is the sprite, and not the commands or expressions which appear within its body. The implementation of Ether was done in Lisp and is very lisp-oriented. This is not inherent in Ether, but is an artifact of its implementation.

1.6 Apiary0

Apiary0 was a design and preliminary implementation of a computer architecture for supporting actor languages [Hewitt 80]. It supports a model of hardware as a large number of physically small processors, each with its own memory, connected by a network of high bandwidth links. Each processor (or *worker*) is independent of the rest, but they cooperate by sending messages to each other over the network.

The Apiary architecture is responsible for providing storage management services. It allocates space for newly-created actors. It also garbage-collects inaccessible actors. A fast, real-time garbage collection algorithm [Lieberman and Hewitt 83] is used for actors existing locally in the worker's memory. A more complex garbage-collection algorithm involving cooperation among workers is used in non-local garbage collection. This algorithm also tries to group related actors onto the same worker, to provide locality of reference, and minimize communications across workers.

The Apiary architecture is responsible for providing computing power for actors. Because it must provide computing power for all actors in its memory, it maintains a queue of tasks. Each task consists of an actor and a communication for it to accept. It dequeues a task and processes it, enqueueing any new tasks which this processing causes.

The Apiary performs reliable transmission of communications to target actors. If the target is on the same worker as the sender, the transmission is fast and trivial, involving only local memory operations. If the target is not on the same worker as the sender, then the worker must communicate with at least one of its neighboring workers, to get the communication on its way to the target. The communication transmission might involve more of this kind activity, depending on how far away the target is, and what forwarding information each worker has. Routing of communications is done dynamically by workers.

The architecture is also responsible for migrating actors and performing load-balancing. Actors can be moved from worker to worker. This is easy for unserialized actors, because they can be copied arbitrarily. Moving serialized actors requires extra synchronization. When a worker's pending task queue is significantly longer than one of its neighbor's queues, the worker can migrate some of the tasks to

its neighbors. In this process of load-balancing, actors are chosen for migration in a fashion that attempts to preserve locality of reference as much as possible, to minimize message-passing across workers.

The Apiary must also be able to deal with physical problems, such as the failure of communication channels or workers.

1.7 Integration

Act1, Omega, Ether, and Apiary0 were experiments dealing with different aspects of the design of a high-level actor-based language system. Work progressed on each, and independent implementations were developed, not all of which were actor-based. An actor language currently being designed is intended to blend the functionality and use the mechanisms of Act1, Omega, and Ether. This language, *Prelude* is expected to run on a computer architecture such as the Apiary, which provide parallel computation facilities, as well as services such as storage management, migration, and transmission of communications.

As an actor language, Prelude will have at least the functionality of Act1. The specific constructs with which it provides that functionality, however, will be oriented more toward the intended usage and flavor of the language. In addition, Prelude will use instance descriptions as information containers and as types. It will use pattern-matching for information extraction, for type-checking, and for recognition of communications. It will also use sponsors for resource management. Pattern-matching with deduction can make use of sprites working together concurrently to establish or deny a relationship between a pattern and the object being matched.

Act1, Omega, and Ether were implemented independently. Each was molded

to deal with specific and well-chosen issues and ideas, and to a large extent ignored the issues dealt with by the others. As a result, their designs and implementations are incompatible. All three are very closely tied to the Lisp language in which they were implemented.

There are also conflicts in their underlying philosophies. Omega and Ether assume monotonicity, and assume that nothing they deal with will ever change. Because of this, they can support parallelism, with no need for synchronization. They can also assume that once something is shown to be true (or false), it will remain that way forever. On the other hand, Act2 has serializers, which can alter their behavior.

In addition, syntactic conflicts arise when attempts are made to integrate the constructs from each with minimal change.

The task of implementing Prelude is factored into the implementation of a few layers. The Lisp language provides an interface to the raw hardware used in the implementation of an Apiary. A language called Scripter which is embedded in Lisp provides an interface to the Apiary architecture, as well as convenient expression of low-level message-passing computations. *Act2* is an actor language implemented in Scripter which integrates the basic mechanisms from Act1, Omega, and Ether, in an extensible fashion. Prelude can be embedded in Act2 with a set of syntactic and semantic extensions written in Act2, by providing the more sophisticated services from Omega and Ether.

Chapter Two

Introductory Examples

The following chapters — especially Chapters Four and Five — assume a familiarity with the Act2 language. This chapter serves as a very brief introduction to the language, by way of illustrative examples. This chapter is merely intended to provide impressions of the language, its constructs and their use. It is not intended to provide a full understanding of the language. Such descriptions have been encapsulated in appendices, for use as reference material. For those who are interested in a deeper understanding of the syntax and semantics of Act2, we recommend browsing through one or more of:

an Act2 tutorial {section B, page 119},
an informal language description {section C, page 124},
or a detailed meta-circular description of Act2 {section D, page 163}.

The choice of appendices should be based on degree of familiarity with Act2 or with previous actor language designs, and on the depth of understanding desired.

2.1 A Simple Recursive Factorial Actor

Our first example is a standard recursive implementation of factorial. The factorial of any integer `n` larger than `0` is the product of `n` and the factorial of `n - 1`. The factorial of `0` is `1`. Factorial is only defined over the domain of whole numbers.

Our recursive implementation of factorial will be a direct realization of the above description. We will establish a definition of a factorial abstraction, so we can then obtain the factorial of a number such as `3` with an expression of the form, `(new factorial (with number 3))`.

A factorial abstraction can be defined with an expression such as the one below, entered as input to an Act2 listen loop.

```
(define (new factorial
            (with number (≡n which-is (a whole-number))))
  (if (= n 0)
    (then do (reply 1))
    (else do (reply (* n (new factorial (with number (- n 1))))))))
```

The **define** expression includes a template describing a particular form of **new** expressions. That is, the template

`(new factorial (with number (≡n which-is (a whole-number))))`

characterizes all **new** expressions like `(new factorial (with number 3000))`, which have a concept which evaluates to the `factorial` concept, and which has a `number` attribute whose filler is a `whole-number`.

The **define** expression also includes an expression which denotes the meaning of **new** expressions described by the template. This expression is evaluated in the environment in which the **define** expression itself was evaluated, extended with any bindings occurring in the template. For example, when an expression such as `(new factorial (with number (+ 2000 1000)))` is asked to evaluate itself in some environment, it looks up the concept `factorial` in the environment.

Next, it asks the pattern

`(a factorial (with number (≡n which-is (a whole-number))))`, which was installed by the **define** expression above, to match an instance description of the form `(a factorial (with number 3000))`. This results in a successful match, binding `n` to `3000`. The definition environment installed by the **define** is extended with this binding, producing a new environment. If the match had failed, a complaint would have been sent immediately as the response to the **new** expression's evaluation request.

Finally, the expression installed by the **define**,

```
(if (= n 0)
  (then do (reply 1))
  (else do (reply (* n (new factorial (with number (- n 1)))))))
```

is asked to evaluate itself in this extended environment. Its response to the evaluation request is sent as the response for the evaluation of the **new** expression itself.

When this expression is asked to evaluate itself, it discovers that `3000`, which is bound to `n`, and `0` do not believe they are equal. In response to the evaluation request, it replies with the product of `3000` and the result of `(new factorial (with number 2999))`.

2.2 A More Concurrent Factorial Actor

In the naive implementation of factorial above, the history of multiplications in obtaining the factorial of 3000 had the form, `(3000 * (2999 * (... * (3 * (2 * (1 * 1)))...)))`. All multiplications had to be performed sequentially, because of the algorithm chosen as the implementation for factorial.

A highly concurrent implementation might view the factorial of 3000 as a product of the integers in the range from 1 through 3000. The algorithm for computing this range product might divide the problem into the product of the range product from 1 through 1500 and the range product from 1501 through 3000. These subproblems are independent, and can be computed concurrently. Moreover, they can be computed in the same manner, spawning even more concurrent activity. Thus, a large number of the multiplications involved in computing the factorial of 3000 could be done concurrently.

Our implementation of factorial can check for the special case of the factorial

of 0, and can make use of a subsidiary **range-product** abstraction for integers larger than 0. Here is the revised implementation of factorial.

```
(define (new factorial
             (with number (≡n which-is (a whole-number))))
  (if (= n 0)
    (then do (reply 1))
    (else do (reply (new range-product
                         (with low 1)
                         (with high n))))))
```

The implementation of the range-product abstraction has special cases, where the lower bound is larger than and where the lower bound is equal to the higher bound. Here is its implementation:

```
(define (new range-product
             (with low (≡lo which-is (a natural-number)))
             (with high (≡hi which-is (a natural-number))))
  (one-of
   (if (= lo hi) do (reply lo))
   (if (> lo hi) do (reply 1))
   (if (< lo hi) do
     (let ((≡mid match (floor (÷ (+ lo hi) 2)))) do
       (reply (* (new range-product
                      (with low 1)
                      (with high mid))
                 (new range-product
                      (with low (+ mid 1))
                      (with high hi))))))))
```

Notice that Act2 expressions such as the multiplication expression, *, are defined to evaluate their arguments concurrently. This is a major source of concurrency in performing range products using this algorithm. Notice also that the **if** branches of the **one-of** expression are tried concurrently. That is, the boolean expressions within them are evaluated concurrently, and the first one (temporally) which is noticed to reply with a **true** value is chosen. The body of the chosen branch is then evaluated.

2.3 A Simple Bank Account Actor

So far, we have seen actors which behave like mathematical functions, performing factorials and range products. These are typical of the programming style espoused by applicative programming aficionados. We can implement other kinds of actors using the same abstraction mechanism. For example, we can define a simple **account** actor, which can represent bank accounts. Our code will be typical of an object-oriented programming style, made popular by Smalltalk [Ingalls 78]. We can establish a suitable meaning for expressions of the form **(new account (with balance 3000))**, so their evaluation results in the creation of new **account** actors with the specified balance.

Here is an example of an implementation of such an **account** abstraction:

```
(define (new account
             (with balance ≡b))
  (create
   (is-request (a balance) do (reply (a balance)))
   (is-request (a deposit (with amount ≡a)) do
     (become (new account (with balance (+ b a))))
     (reply (a deposit-receipt (with amount a))))
   (is-request (a withdrawal (with amount ≡a)) do
     (let ((≡new-balance match (- b a))) do
       (if (≥ new-balance 0)
         (then do
           (become (new account (with balance new-balance)))
           (reply (a withdrawal-receipt (with amount a))))
         (else do
           (complain (an overdraft))))))))
```

When an expression of the form **(new account (with balance 3000))** is asked to evaluate itself, it behaves in much the same way as **(new factorial (with number 3000))**. The difference is that the expression which gets evaluated is a **create** expression, which represents the creation of an actor whose behavior is described by the *communication handlers* in the **create** expression. These communication handlers classify the communications which the actor can accept for processing, and describe what the actor will do to process each

communication. For example, the first communication handler is for requests containing a message which match `(a balance)`. The actor replies to such a request with a reply containing the current balance as its message.

The second communication handler is for requests containing deposits. When such a request is received, the account concurrently replaces itself with a new account with an appropriately increased balanced, and replies with a deposit receipt for the deposited amount.

The third communication handler is for requests containing withdrawals. When such a request is received, the account must first check for an attempt to withdraw more than the current balance. If this happens, then it complains with an overdraft and does not alter its behavior. If the amount is a valid one, the account concurrently replaces its behavior with an appropriately decreased balance, and replies with a receipt for the withdrawal.

Notice that the **become** and **reply** commands are evaluated concurrently in the environment in which the actor was created, extended with the bindings of local variables in the communication handlers and enclosing commands. Note also that when the account receives a communication, the communication handlers attempt to match it concurrently.

2.4 A New Control Abstraction

As a final example, we will define abstract syntax for an expression with which we could extend the language. On the surface, this expression might look like `(first-response` *exp1 exp2*`)`. We would like the evaluation of this expression to respond with the first response it gets when it concurrently asks the two expressions to evaluate themselves. As soon as it relays the first response, it should stop

sponsoring the second computation and should discard the second response.

Our implementation of the `first-response-expression` abstract syntax actor will make use of two subsidiary abstractions, an `initial-sponsor` and a `subsequent-sponsor`. The strategy is to collect all relevant information present in the `expression-eval` request, including the message, customer, complaint--department and sponsor. We create an `initial-sponsor` actor using the original customer, complaint-department and sponsor. We then send both sub-expressions a request containing the original `expression-eval` message, but designate the `initial-sponsor` actor we just created as the customer, complaint-department and sponsor.

```
(define (new first-response-expression
             (with expression-1 ≡exp1)
             (with expression-2 ≡exp2))
 (create-unserialized
  (is-communication
   (a request
      (with message (≡original-message which-is (an expression-eval)))
      (with customer ≡original-customer)
      (with complaint-department ≡original-complaint-department)
      (with sponsor ≡original-sponsor))
   do
   (let ((≡is match
           (new initial-sponsor
                (with customer original-customer)
                (with complaint-department
                      original-complaint-department)
                (with sponsor original-sponsor))))
     do
     (let ((≡new-expression-eval match
             (new request
                  (with message original-message)
                  (with customer is)
                  (with complaint-department is)
                  (with sponsor is))))
       do
       (send-to exp1 new-expression-eval)
       (send-to exp2 new-expression-eval))))))
```

The `initial-sponsor` actor is a serializer which serves as a sponsor for the evaluation of the two sub-expressions, as a customer for collecting replies, and as a

complaint-department for collecting complaints. As a sponsor, it should relay any requests for more resources to the original sponsor. As a customer and complaint department, it should relay the first response to the original customer or complaint department, as appropriate. As it does this, it should also become a **subsequent-sponsor** actor, which will refuse to grant more resources and will discard any other response.

```
(define (new initial-sponsor
             (with customer ≡c)
             (with complaint-department ≡cd)
             (with sponsor ≡s))
  (create
   (is-request (≡message which-is (a resource-request)) do
     (reply (ask s message)))
   (is-reply ≡message do
     (reply-to c message)
     (become (new subsequent-sponsor)))
   (is-complaint ≡message do
     (complain-to cd message)
     (become (new subsequent-sponsor)))))
```

The implementation of the **subsequent-sponsor** abstraction is quite simple. It complains when asked for more resources, and does nothing in response to any replies or complaints it receives.

```
(define (new subsequent-sponsor)
  (create-unserialized
   (is-request (a resource-request) do
     (complain (a no-resources-available)))
   (is-reply something do )
   (is-complaint something do )))
```

Chapter Three

The History of Act2

One part of this thesis work has been work on the design of Act2. The design effort consisted of taking the preliminary design for Prelude itself as documented in [Theriault 82], analyzing it with respect to our design goals, self-consistency, uniformity, and implementability, and making modifications as necessary. Some changes were made less to the syntax of constructs than to their semantics. The design of Act2 involved evaluating the preliminary, documented design, checking for consistency, synergy, simplicity; evaluating them in terms of new design goals and principles; deciding what could be factored out into a base language and what could be embedded in this language. Integration was envisioned in the preliminary design, but its details had not been worked through. Some forms of bottoming out had been addressed by Lieberman's Act1 implementation, but bottoming out of scripts and instance descriptions was peculiar to the requirements for Act2.

As design began, so did the beginnings of an implementation, in order to further develop intuitions for how much work is done in message-passing using instance description, for the problem of bottoming out, and for what implementation aids would be useful. At this time, Scripter did not yet exist, and an Apiary simulator for the Lisp Machine was still in its infancy. An implementation in Lisp would have been very bulky, time-consuming, and difficult to read and modify. The circularity problems in bottoming out Act2 are more acute than they were for Act1, and this would have accounted for significantly more code.

We decided to write a meta-circular description of Act2, using it as a tool in

the design of Act2 itself. The meta-circular description, being an abstract implementation of Act2, also provided an opportunity to plan and experiment with implementation strategies.

Once the language design had settled to a reasonable extent, Scripter and the Apiary simulator were beginning to become usable for small experiments. We decided to implement expressions for a small toy language, as if they were part of Act2 itself. An implementation for these expressions was first written in Act2, to demonstrate its generality, flexibility, and readability. This included the implementation of an actor-based listen-loop, event-based parsing, and event-based evaluation.

Next, an implementation for the toy language was attempted in Scripter, to provide higher-level testing of it and to point out any problems and deficiencies in the interface it provided to the Apiary. Once the fundamental portions of the toy language had been implemented, progressive extensions were made to it, to work out more of the implementation problem, including bottoming-out of primitive actors and implementing serializers.

This set the stage for an implementation of Act2 in Scripter. The next step might have been to integrate descriptions and pattern-matching into the toy language. This was a quantum leap in the complexity and size of the language. Instead, work was started on the implementation of a rudimentary version of Act2 in Scripter. This grew into the present implementation of an Act2 interpreter.

3.1 A Meta-Circular Description of Act2

3.1.1 Perspective

The meta-circular description is best understood by first understanding the context in which it exists. A user's interface to Act2 is an event-based listen-loop, with an operating environment in which names are resolved. The listener first accepts input from the user in the form of list structure, symbols, and numbers. It asks this input to parse itself, producing an actor which represent the abstract syntax of the input. This actor may have acquaintances which represent the abstract syntax of portions of the input. The listener then asks the abstract syntax actor to evaluate itself as an expression in the current environment.

Each abstract syntax actor is responsible for its own evaluation. Rather than having a single interpreter, which accepts, parses, and evaluates the input, Act2's approach is "actor-based" or "object-oriented." The interpretation process is a cooperative one, with knowledge about each construct localized in the implementation of the construct.

An interpreter for Act2 consists of a set of actors which parse list structure into abstract syntax objects, and abstract syntax objects which evaluate themselves and create actors or transmit communications as appropriate. Our meta-circular description consists of an Act2 implementation of abstract syntax objects representing Act2 constructs. That is, we describe the processing which occurs when the abstract syntax object receives a request to evaluate itself in some environment.

The meta-circular description provides a form of informal, high-level operational specification of the semantics of each construct. Because of the circularities which naturally arise in an Act2 description of itself, our meta-circular

description is mathematically vacuous. It does, however, convey to its reader a fairly accurate idea of just what each construct means, in a relatively clear, concise, and precise manner. This made it useful for discussing the design decisions and problems with others. It was often less ambiguous than corresponding English descriptions.

It was also useful because of the way it allowed us to postpone dealing with low-level implementation detail, such as exactly how communications are transmitted, how actors are implemented, how Act2 bottoms out into and interfaces with the underlying architecture. Rather, it distills out the high-level problems and issues, so they can be dealt with directly, rather than indirectly by debugging a large and detailed implementation. For the same reason, it increased the likelihood of experimenting with alternatives, because they were relatively quick and easy to try out. In the long run, this saved much time and implementation effort.

Because the meta-circular description was written in a programming language, it made case analysis more natural. The likeness to programming tended to promote completeness and attention to detail. Often, troublesome cases which might otherwise have been ignored or taken for granted became apparent. This also allowed us to use the programming intuitions, which we have acquired through implementation experience, in the design process.

In addition, writing the description of Act2 in Act2 provided us with intuitions about what Act2 programming would be like, and what Act2 code would look like. This experience in itself was responsible for a few changes. Implementing Act2 in itself also demonstrates its generality as a programming language.

In hindsight, the meta-circular description was a very useful design and implementation tool. The structure and content of the Scripter implementation of

Act2 was modeled closely after the meta-circular description, and progressed smoothly as a result.

3.2 A Toy Language Implementation Experiment

When our meta-circular description had become relatively stabilized, we began to experiment with the implementation of a very simple expressional language. Part of our purpose was to specify the expressions in the language as syntactic and semantic extensions to Act2. Since no implementation of Act2 existed, the implementation of our toy language would actually be implemented as if part of Act2, requiring only the additional implementation of an event-based listen-loop, implementation of environments, and installation of appropriate behavior for numbers, symbols and lists.

Initially, we needed only unserialized actors, which was fortunate, since our apiary simulator did not support serializers. Our environments were unserialized, even though we realized they would eventually need to be serialized. The expressions we chose to start with were representative of the lambda calculus. A lambda expression provides the ability to lambda-abstract an expression with respect to an identifier. Any free identifiers in the expression are statically bound. When evaluated, a lambda expression replies with a unary operator. When this operator is "applied" to an operand, the expression it abstracted from is evaluated in its original context, but with the lambda-variable bound to the operand. Such an application in our actor-based design consists of sending the operand as a message in a request to the operator.

Our **lambda** and **apply** expressions have the form:

(lambda *lambda-variable-symbol abstracted-expression*)
(apply *operator-expression operand-expression*)

An implementation in Act2 was trivial. The **apply** expression simply evaluates the operator and operand expressions, then sends the evaluated operand (wrapped in a request) to the evaluated operator. The Act2 code is presented below simply for illustrative purposes, to present an image of the language, its use and expressiveness. Code in this section is intended mainly to provide imagery, and its details need not be understood except by readers who are interested enough to browse through language descriptions in the appendices.

```
(define (new APPLY-EXPRESSION
             (with operator ≡op)
             (with operand ≡x))
  (create-unserialized
    (is-request (≡eval which-is (an expression-eval)) do
      (reply  (ask (ask op eval) (ask x eval))))))
```

The **lambda** expression simply results in a closure, which retains the variable, expression, and environment for later use as an operator.

```
(define (new LAMBDA-EXPRESSION
             (with variable ≡var)
             (with body ≡exp))
  (create-unserialized
    (is-request (an expression-eval (with environment ≡env))) do
      (reply (new closure
                  (with variable var)
                  (with body exp)
                  (with environment env))))))

(define (new CLOSURE
             (with variable ≡var)
             (with body ≡exp)
             (with environment ≡env))
  (create-unserialized
    (is-request ≡val do
      (reply
        (ask exp
          (an expression-eval
              (with environment
                    (new environment
                         (with primary
                               (ask (new empty-layer)
                                    (a grow
                                       (with symbol var)
                                       (with value val))))
                         (with secondary env)))))))))
```

The language also had some simple expressions, to facilitate experimentation. Numbers evaluated to themselves. The symbols **true** and **false** were bound in the initial environment to primitive actors with appropriate behaviors.

With the addition of an **if** expression, to choose between two expressions to evaluate, the language had the ability to make decisions. The if expression had the form: (**if** *boolean-expression expression-if-true expression-if-false*).

Given this as a base, we demonstrated that Act2 indeed had the expressive power to implement the lambda calculus, and the elegance to implement it in simple, readable code. We also wrote Act2 code implementing environments, and representing the behavior of numbers, symbols, and lists.

This established, we set about implementing environments and a listen-loop in Scripter. We provided a scripter interface for Act2 to customize the behavior of primitive actors. We implemented event-based parsers for the constructs, and installed them in an expression-parsing environment. We implemented abstract syntax for each expression, which knew how to evaluate itself, given an environment. We ran experiments on the apiary simulator, entering expressions in our experimental language, noticing what they parsed and evaluated into, and noticing how many events were required for parsing and evaluation. Printing of actors was done by Lisp functions.

Some logical and numeric expressions were provided, to express simple computations. These provided somewhat larger and more interesting test cases.

(**not** *boolean-expression*)
(**and** *boolean-expression boolean-expression*)
(**or** *boolean-expression boolean-expression*)
(**eq** *expression expression*)
(**+** *numeric-expression numeric-expression*)
(**-** *numeric-expression numeric-expression*)
(***** *numeric-expression numeric-expression*)

We found it desirable to further extend our toy language, to remember the results of previous computations. We invented a construct for extending the loop's prevailing environment by binding a symbol to the result of evaluating an expression. In order to make this work right, we introduced a simple implementation of serializers to the Apiary, to provide serialized environments.

Our new construct had the form: **(defname** *symbol expression*).

It allowed us to construct recursive operators, simply by entering an expression such as:

```
(defname factorial
  (lambda x
    (if (eq x 0)
        1
        (* x (apply factorial (- x 1))))))
```

Our implementation of expressions requiring the evaluation of sub-expressions was a simple one. It would evaluate the sub-expressions sequentially from left to right, obtaining the result from the leftmost before beginning the evaluation of those to the right. Some evaluators for the lambda calculus have included mechanisms for lazy or eager evaluation. For example, in an expression such as **(apply (lambda x 3)** *operand-expression*), it is not necessary to evaluate the *operand-expression* because it is not used in the body of the lambda expression. Also, in an expression such as **(apply (lambda x (+ x x))** *operand-expression*), the lambda calculus' substitution semantics would evaluate *operand-expression* twice. Introduction of lazy evaluation mechanisms to lambda calculus interpreters prevents unnecessary or duplicate evaluations of expressions such as these.

Lazy evaluation is easy to add to our little language implemented in Act2. We can simply extend Act2 to include a simple **delay** expression, which replies immediately with an actor. This actor saves the evaluation environment and the

expression's abstract syntax. If no message is ever sent to the delay, it never evaluates the expression. If any are sent, the delay evaluates the expression, replaces itself with the result, then processes the incoming communication. The delay expression can have the form **(delay** *expression*) and can be implemented in Act2 as:

```
(define (new DELAY-EXPRESSION (with expression ≡exp))
  (create-unserialized
    (is-request (≡eval which-is (an expression-eval)) do
      (reply
        (create
          (is-communication ≡c do
            (let ((≡value match (ask exp eval))) do
              (send-to value c)
              (become value)))))))
```

We can selectively denote the lazy evaluation of an expression by explicitly saying **(delay** *expression*). For example, a programmer can guarantee lazy evaluation of operands by writing **apply** expressions like **(apply (lambda x 3) (delay** *operand-expression*)). In this case, the *operand-expression* would never be evaluated, because the operator would simply reply with the value, **3**.

Alternatively, we can have all operands to apply expressions be evaluated lazily by trivially modifying our implementation of the apply expression:

```
(define (new APPLY-EXPRESSION
             (with operator ≡op)
             (with operand ≡x))
  (create-unserialized
    (is-request (≡eval which-is (an expression-eval)) do
      (reply  (ask (ask op eval) (delay (ask x eval)))))))
```

The ability to implement the **delay** and **hurry** expressions required a full implementation of serializers in the Apiary. Eager evaluation was implemented using futures. It was expressed in our language as **(hurry** *expression*). When evaluated, a hurry expression immediately returns with a *future* actor which

represents the result of the evaluation, in much the same way that a delay actor did above. However, it concurrently asks the expression to evaluate itself, and to respond to the future. Until it receives the response, the future will enqueue communications intended for the value. Once it obtains the value, the future will become the value and will also send all of the enqueued communications to it for processing. The implementation of futures is a bit more complicated than the implementation of delays. Note that the response from the evaluation of the expression must be distinguished by the future from communications sent to the value. We provide this ability by using the authentication mechanisms provided by Act2.

Given the existence of the **hurry** expression, we can explicitly denote eager evaluation of an expression with **(hurry** *expression*).

We can provide eager evaluation by default in our little language by modifying the apply expression, so the evaluation of the operation proceeds concurrently with the evaluation of the operand.

```
(define (new APPLY-EXPRESSION
             (with operator ≡op)
             (with operand ≡x))
  (create-unserialized
    (is-request (≡eval which-is (an expression-eval)) do
      (reply  (ask (hurry (ask op eval)) (hurry (ask x eval)))))))
```

In the evaluation of an expression such as **(apply (apply op arg1) arg2)**, the expressions, **op**, **arg1**, and **arg2**, are evaluated concurrently.

Some researchers [Backus 78, Dennis 81, Turner 79] believe applicative languages to be ideal for concurrent programming. Because every expression is completely functional, and has no side-effects, the order in which expressions are evaluated is irrelevant. They typically introduce eager evaluation into interpreters for these languages, in order to realize this potential for concurrency. No matter

how they implement their interpreters, some amount of synchronization is necessary. In general, this synchronization requires the notion of state change. Because of this, the applicative languages are not powerful enough to implement their own interpreters. Similarly, these languages are not powerful enough to implement interpreters with lazy evaluation.

3.2.1 Act2 Implementation

The implementation of Act2 in Act2 has the same style as the implementation of the toy language expressions in Act2. It does, however, handle complaints wherever they may occur. The listen-loop interface to Act2 has event-based, object-oriented parsing {section 5.2, page 96} which makes the language extensible. Abstract syntax objects representing Act2 expressions and commands are responsible for their own evaluation.

Making syntactic extensions to Act2 is relatively simple. A programmer simply extends the appropriate expression or command parsing environment, mapping some symbol which will serve as a keyword to a user-supplied parser. This parser will parse list structure denoting an instance of the construct into some abstract syntax actor. Act2 provides a construct for establishing this in a simple way. For example, here is how we might establish a connection between the concrete and abstract syntax for **apply** and **lambda** expressions:

```
(defexpression apply
  ;; a parser for "(apply EXPRESSION OPERAND)"
  (create
   (is-request (an expression-parse (with source ≡src)
                                    (with expression-keywords ≡ek)
                                    (with command-keywords ≡ck))
     do
     (case-for src      ;; note that a LIST is a (simple) SEQUENCE.
       (is ['apply ≡op ≡arg] do
         (reply
          (new application-expression
               (with operator
                     (ask op (a parse-yourself-as-expression
                                (with expression-keywords ek)
                                (with command-keywords ck))))
               (with argument
                     (ask arg (a parse-yourself-as-expression
                                 (with expression-keywords ek)
                                 (with command-keywords ck))))))))))))
```

It is quite likely that installations of Act2 will provide generalized parser abstractions, which will often eliminate the need to write code like that shown above. Assuming the existence of such an abstraction, **prefix-parser**, the installation of the **lambda** expression might look like:

```
(defexpression lambda
  (new prefix-parser
       (with keyword 'lambda)
       (with number-of-arguments 2)))
```

Act2 is also semantically extensible, because a user may define his own abstract syntax objects, or redefine pre-existing ones. The implementation of Act2 constructs consists of definitions of appropriate parsing and abstract syntax actors, and the definition of any actors which are useful to create dynamically.

The implementation of Act2 in Scripter is closely patterned after the meta-circular description. It uses the same major implementation strategies. It differs in the details, because Scripter does not have the expressiveness of Act2.

Chapter Four

Issues in the Design of Act2

4.1 Act2 is Part of a Layered Implementation

Act2 is part of a layered approach to the design and implementation of a more sophisticated actor language system. Prelude is intended to incorporate and augment the functionality of the Act1, Omega, and Ether experiments. It will integrate their fundamental mechanisms and higher-level approaches, ironing out their differences in philosophy. The result will be a high-level, highly-concurrent programming language with knowledge representation and problem solving capabilities. This language system will be accountable for the actors implemented and created within it, making possible the cooperation of applications written within it with independently-conceived application systems.

The design and implementation of Prelude is a rather ambitious project. Attempting to implement it all at once would very likely lead to difficulties, as the implementation of Plasma did, and might result in a very bulky implementation which was difficult to understand and evolve.

Act2 was designed to serve as a substrate for the implementation of Prelude. It can also stand as a programming language in its own right. Act2 addresses practical issues involved in an interface with the computer architecture below. It addresses issues involved in an interface for programming applications and embedding languages above. In addition, it addresses issues involved in integrating the fundamental mechanisms of Act1, Omega, and Ether into a coherent language base. In this way, a substantial set of issues can be addressed by a manageable

project, without incurring the burden of a full-scale implementation of Prelude.

Prelude's additional functionality can be embedded in Act2, using Act2 implementation mechanisms, without the need to address those issues already addressed by Act2. Additional features include the ability to construct and manipulate lattices of descriptions, related by inheritance; the ability to make deductions based on these relationships; more sophisticated resource management policies; and dissemination of information.

4.1.1 Act2 Assumptions

Act2 itself is built on top of other layers, in which other sets of problems are factored out and solved. Lisp-Machine Lisp [Weinreb and Moon 81] provides a comfortable interface to the underlying hardware, and provides abstractions suitable for representing actors. The worker interface to the Apiary architecture is implemented in Lisp, and provides a view of the underlying hardware as part of an actor-based apiary. Scripter is a macro language embedded in Lisp, which provides a high-level interface to workers, allowing computations to be expressed in terms of actor creation, one and two-way communication, and change of behavior, in addition to Lisp code. Act2 is implemented in Scripter.

An important part of the design and implementation of Act2 is an assumption about the character of computation on the underlying computer architecture. For example, Act2 assumes that the transmission of communications is reliable, cheap, and quick. The worker optimizes the transmission of a communication when the target is on the same worker as the sender. Workers attempt to maintain locality of reference when migrating actors to other workers.

Act2 assumes that creation of actors is very cheap. All that is involved in actor

creation is the allocation in Lisp of a small data structure to represent the actor.

Act2 assumes that actors are garbage-collected, and that the garbage collection algorithms are efficient and effective. Each worker does local real-time garbage collection of inaccessible actors. An algorithm has been developed for this, which reclaims storage quickly [Lieberman and Hewitt 83]. Garbage collection across workers is a more difficult problem, requiring a more sophisticated algorithm by which workers cooperate to localize inaccessible actors for intra-worker reclamation.

Act2 assumes memory is inexpensive and plentiful. This is becoming more and more true with time. Act2 also assumes that copying and maintaining multiple copies of unserialized actors on different processors is cheap. Actors which cannot change can be copied indiscriminately, to increase locality of reference, and to make migration easier.

Act2 assumes that the underlying computer architecture may consist of large numbers of processors interconnected by high-bandwidth links. The apiary was designed with this in mind. Workers perform load-balancing and migration, in order to make use of the available parallelism.

A consequence of these assumptions is that a high degree of concurrency may be obtained by expressing computations in terms of large collections of highly specialized actors communicating with each other by transmission of communications. That is exactly the style of computation supported by Act2.

4.1.2 Act2 Design Goals

4.1.2.1 Integration as a Design Goal

Act2 is designed to integrate the fundamental mechanisms developed in experiments with Act1, Omega, and Ether. As such, it is an actor-based language, founded on message-passing semantics. It provides mechanisms for creating actors, for point-to-point communication, and for expressing two-way communication.

Act2 makes use of descriptions in fundamental and pervasive ways, which allow for them to coexist with other actors. It implements its own mechanisms for pattern-matching, which do not involve deduction. Inheritance and deduction mechanisms can be introduced as extensions to the language.

Act2 uses sponsors as its fundamental resource-management mechanism for controlling asynchronous computations. It is possible to implement sprites as actors, and to introduce more sophisticated resource management policies. Sprites can work off patterns which are descriptions, and dissemination of information can be performed in coordination with description lattices and point-to-point communication.

4.1.2.2 Expressive Power as a Design Goal

Expressive power is an objective measure of the generality of a language. Because it will be used to implement Prelude and arbitrary applications for concurrent systems, Act2 must be general enough to express whatever might be necessary. Generality includes the ability to deal with concurrent systems, including those which do not assume a closed world model.

One criterion for generality is the ability to implement Act2 in itself. The meta-circular description {section D, page 163} is evidence of this. Another is the direct support for the actor model, whose generality has been considered

independently.

Act2 is expected to be used to implement languages as well as applications. For this reason, it needs abstraction mechanisms and mechanisms for syntactic extension.

In addition, Act2 is expected to deal with issues such as protection, security, and authentication, to protect the integrity of actor systems and allow controlled sharing of information.

4.1.2.3 Expressiveness as a Design Goal

Expressiveness is a highly subjective measure of the quality of a language. It involves such areas as simplicity, friendliness, benevolent character, and range of application of a language. For example, Act2 was designed to be interactive. Act2 was not designed to be a minimal language, providing only enough mechanism for the integration and generality goals. Instead, its constructs are geared more toward understandability and programmability. It includes software engineering features in addition to those of Act1. Act2 was not designed to provide everything a programmer might want, but to make it possible and convenient to embed further and more useful mechanisms.

4.2 Programmer Interaction

One of the important aspects of a language is the interface it presents to a programmer. We take a broad notion of "programmer" to include both traditional human programmers and computer programs which write or manipulate other programs. There are many trade-offs in user-interface design, and more are introduced by this broad concept of a user.

Our general approach in the design of Act2 has been to attempt to maximize flexibility and generality. Specifically, in the area of programmer interface, we have opted for a language which is highly interactive in character, for comfort in programming. We have also attempted to decouple issues of syntax and semantics, so they could be handled separately.

4.2.1 Interactiveness

One of the basic requirements for Act2 was that it be designed to be interactive in nature. [Sandewall 80] demonstrates the utility of interactive programming environments. Experience has shown that more interactive programming environments tend to be more comfortable to work with and provide a friendlier human interface to the machine and language system [Algol, Lisp, Smalltalk, Halbert-thesis]. Act2 was designed as an interpretive language; compilation is treated as an optimization which is engineered to fit within this framework. The interface to Act2 is a listen-loop, similar in nature to that of Lisp, which accepts as input any expression in the language. This encourages a more conversational interaction between man and machine.

There is always an environment associated with the listener for resolving symbols used in the user's input. This environment corresponds both to the context of a conversation and to a personal data-base. The environment is preserved from session to session, to provide a sense of continuity.

Special expressions exist for the binding of names and definition of abstractions at the top level. They alter the semantic content of the prevailing environment, in order to preserve the definitions for later use. A user can associate names with specific actors using the **defname** expression, and can can define abstractions using the **define** expression. Most other expressions in the language do

not affect the environment in which they are evaluated.

4.2.1.1 Actor-Based Interpretation

Act2 is implemented in the same style we advocate for all applications. There is no centralized interpreter for the language. Instead, each construct is implemented by an abstract syntax actor, which is responsible for its own evaluation. This makes semantic extensions possible in a very natural manner — we can simply define new abstract syntax actors using the same mechanism for implementing any other abstraction. The implementor only needs to make the abstraction obey the communication protocols of abstract syntax actors, accepting communications such as requests for evaluation or compilation.

Parsing is also actor-based. Each construct parses itself, using a parser which has been associated with the construct. The listener reads in user input as a set of nested syntactic phrases, represented as a composition of list structure, symbols, and numbers. Each syntactic phrase is asked to parse itself. List structure scans itself, looking for a symbol which has been defined as a keyword for some construct. It then delegates the job of parsing to the parser which has been associated with that keyword.

This method of parsing makes syntactic extension of Act2 an easy matter. A programmer can install a new keyword/parser pair using the **defexpression** and **defcommand** expressions. Such declarations are done at top level, to the listener. Once again, the actor-based programming discipline gives us the flexibility we desire.

4.2.2 Act2 Separates Syntax from Semantics

The actor-based implementation of Act2 decouples the activities of parsing and evaluation of language constructs. In so doing, it provides natural means for syntactic and semantic extension of the language. In addition, it decouples the syntax of the language from its semantics. A set of abstract syntax actor definitions embody the semantics of the Act2 language.

These abstract syntax actors are largely independent from the concrete syntax which is mapped onto them by a set of parsers. This separation of syntax and semantics allows a large degree of separation of style from mechanism, of presentation from representation, of form from function, and of syntactic issues from semantic issues. It allows us as language designers to concentrate on different sets of issues separately.

We took advantage of this by concentrating on semantic issues and requirements. We chose a concrete syntax which closely resembles the abstract syntax we found desirable. Alternative sets of constructs can be mapped onto this set of abstract syntax actors if and when desired.

4.2.2.1 Presentation and Editing Tools

We gain additional benefits from this decoupling of syntax from semantics. Presentation tools can operate with abstract syntax objects, and provide alternative ways of looking at them, based on such things as programming style, familiarity of the reader with the code, indentation preferences, and available space. This can provide a more comfortable way to read code written by others.

Editing tools can make it more comfortable to write Act2 code. An editor can provide templates for the programmer to fill, decreasing the amount of typing

needed. It can also allow programmers the luxury of personal short-hand, which it converts to the appropriate abstract syntax.

4.2.3 Syntactic Issues

4.2.3.1 Bracketed Syntax

Act2 has a bracketed syntax. This was chosen because we needed a convenient, uniform way of recognizing phrases and sub-phrases in the language. It provides us with this ability even in the face of arbitrary syntactic extensions It makes the language amenable to convenient construction and analysis by computer programs as well as human programmers. It allows lexical analysis to be performed automatically, and to be ignored by those making extensions. This in itself removes a large part of the complexity of parsing. It reflects the structure of computer languages in their syntactic representations. This makes the problems of parsing and extension tractable.

User input is read in as nested list structure, grounded by "atomic" tokens such as symbols and numbers. Each list, is asked to parse itself. In doing so, it scans itself for a symbol for which has been established a keyword/parser pair. This is the mechanism by which syntactic extension is made possible and practical in Act2. Bracketed syntax is the most natural way we know of to provide these capabilities.

4.2.3.2 Template English

In choosing a concrete syntax for Act2, one of the guidelines we used was to try to make it resemble English as much as possible. Whenever possible, we attempted to make the meaning of constructs closely resemble the intuitive meaning of the phrases denoting them, giving Act2 an air of familiarity and understandability

even to novice readers. An Act2 construct often reads somewhat like text, with the addition of parentheses to mark off essential clauses. This is especially noticeable in instance descriptions and **new** expressions. It is also evident in more complex expressions, such as **case-for**, **if**, **one-of**, and **let**. Some compromises were made for the sake of conciseness. There is a point where verbosity ceases to enhance readability because a sense of structure is lost. We believe Act2 strikes a good balance, being verbose enough to be relatively understandable to novice readers, using parenthetical template English to make the structure of the code visible, and avoiding overly verbose concrete syntax for constructs.

Programs are also made more readable by the existence of more familiar expressions of common primitive operations. For example, `(+ 3 5)` can be used instead of `(ask 3 (a + (with operand 5)))` In addition, programmers can use the infix notation `(3 + 5)` if they feel more comfortable with it. This is slightly more readable for novices and has more of an English-like "flow" to it. Unfortunately, with the opportunity for arbitrary syntactic extension, there is a danger of confusing leading identifiers with keywords. For example, the identifier `a` in `(a + b)` might cause confusion when parsing the expression, because it resembles an instance description. For this reason, Act2 warns programmers when they attempt to bind an identifier which also happens to be an expression keyword.

4.2.3.3 Verbosity

One important trade-off in Act2 syntax is verbosity. On the one hand, a language which is overly verbose may be cumbersome to write programs in, and may even be less readable if the main ideas and algorithms are lost in words and symbols. On the other hand, a language like APL which is overly terse can be very cumbersome to read, even for those who have made the effort it takes to become fluent in it. The bias in Act2 is toward readability, at the expense of increased

verbosity. Our assumptions are that more code is read than is actually written or modified, and that the readers will often not be the original writers. It attempts to combine the local-understandability benefits of natural language phrases with the more global-understandability benefits of structured code. Of course, Act2 is somewhat flexible about the whole matter, allowing programmers to introduce more concise or verbose forms of constructs, using the syntax extension mechanisms. Modern editors have abbreviation facilities and other writing aids. Some deal directly with the syntax of a language. With the similar tools, an Act2 programmer should have few reservations about writing "verbose" code.

If we look at Act2 code more closely, we find it difficult to justify a more concise syntax for its constructs. We could shorten the keywords, but the language would become more cryptic. Instance descriptions are about as concise as they can be, without an adverse effect on readability. As they are now, they read just like English text and has exactly the connotations we intend, enhancing the imagery of even experienced readers. **New** expressions could be forsaken in favor of a positional notation, but we would lose the value of keywords, which are a great aid to readability and understandability. One major source of bugs in the history of Lisp programming has been interface problems and misunderstandings, because Reading code is difficult without flipping back and forth between function calls and definitions, to see what each parameter means. In addition, the strong resemblance between **new** expressions and instance descriptions is strongly suggestive of the relationship between actors and their descriptions, and of Act2's flexible notion of instantiation of abstractions.

Act2 constructs have been designed so their most common usages are also their most concise. For example, the **otherwise** clauses in **create**, **one-of**, and **case-for** constructs are rarely needed, and can simply be omitted. The usual intent of programmers is to simply complain if none of the possibilities they allowed for

actually occur. In addition, all constructs relay any unhandled complaints which might occur in the evaluation of sub-expressions within them, eliminating a need to explicitly wrap a handler around the sub-expressions.

There are many cases in Act2 where the programmer is allowed not to explicitly denote information which can be derived from context. Commands like **reply** and **complain** allow the programmer not to mention the intended target, when handling a request communication. The **become** command refers to the enclosing serializer, which need not be explicitly mentioned. When the replacement actor is simply another instance of the same abstraction, the **new** expression within the **become** command need only mention those attributes which will be different. In general, Act2's constructs behave in such a manner that customers, complaint departments, and sponsors need not be explicitly mentioned by programmers.

4.2.3.4 Keyword-Based versus Positional Instantiation

It is possible for Act2 code to be presented in a more condensed form, when desired. **new** expressions can be presented with a lisp-like function call notation which eliminates keywords. Programmers can easily make an extension for a smalltalk-like keyword notation.

There are serious issues to consider when choosing a style. From a software engineering standpoint, it is very useful for a keyword to describe the significance of each parameter of an instantiation. The attribute relations in **new** expressions are very useful for this purpose. They serve as good documentation for readers, and allow extra consistency checking between the instantiation and the definition. They also eliminate the problems which occur when parameters are permuted. The main advantage of positional notation is its conciseness. In writing a program, it is very convenient to reduce typing, and in reading a program, it sometimes makes the

overall algorithm more apparent by reducing the amount of text required to represent it. The benefits of positional notation are often easily obtained with appropriate editing and presentation tools. Act2 provides programmers with the ability to choose which style they wish for each individual instantiation.

4.2.3.5 Extensibility

Part of the success of Lisp has been its extensibility. This feature allowed other languages to be embedded within it. It also allowed the language itself to grow to include increasingly sophisticated and useful features.

We also wanted Act-2 to be syntactically extensible, for these and additional reasons. We feel it may be desirable to develop more than one concrete syntax for Act-2, to serve the needs, desires, and customs of programmers with different styles. Syntactic extension allows programmers to choose a level of verbosity which best serves their needs, and to introduce whatever syntactic sugaring they wish into the language. Customization is an important property of a language which is to be used by disparate institutions.

Assuming syntactic extensibility allowed the Act2 language design to go on at the abstract syntax level, without much concern for the syntactic details. It also allowed us, in the language design phase, to choose a concrete syntax which is very near the abstract syntax, permitting us to concentrate independently on underlying mechanisms and programmability.

Providing the ability for embedding Prelude in Act2 saves us from a full implementation of Prelude; instead, we only need to program the extensions. It is difficult to anticipate now the syntactic and semantic requirements of Prelude, so syntactic extension is even more important.

4.2.4 The Expressive Character of Act2

There were some guidelines we used while choosing an abstract syntax for Act2. Many of the decisions which needed to be made were very subjective in nature, dealing more with expressiveness than with expressive power. We did not intend for Act2 to be a kernel language for implementing Prelude. Instead, we wanted it to be a full-fledged programming language in its own right, with emphasis on mechanisms for good software engineering. This is necessary because the implementation of Prelude is a rather complex task in itself, and should be done with a suitably high-level and comfortable language. These criteria were deemed more important than the size of Act2 and the complexity of its implementation. As a consequence, Act2 has high-level, very flexible constructs, such as **create**, **case-for**, **one-of**, and **let**.

4.2.4.1 Familiarity

One of the guidelines we followed was to make Act2 syntax be as similar as possible to familiar syntax. The syntax for instance descriptions and patterns were borrowed, unchanged, from Omega. Our notation for instantiation of abstractions is almost identical to the notation for instance descriptions, to make them readable, and to suggest a close relationship between the two ideas. Whenever possible, we attempted to use the syntax described in [Theriault 82]. Above all, we did not want to make the language much more complex to read or work with. We made an effort to express familiar ideas and constructs in familiar ways and with commonly-understood notations. For example, we permit the use of infix notation in expressions.

4.2.4.2 Economy of Concept

There are relatively few fundamental concepts in Act2. All computation is ultimately expressed in terms of actor creation and replacement, communication transmission, and simple decision. Properties of actors and the actor model are exploited in the language, to avoid introducing new concepts and constructs. There is also the familiar and intuitively appealing notion of description. They are used as information containers as well as "types" in the language. Pattern matching is used for recognizing and extracting information, binding names, accepting communications, handling complaints, dispatching on values, testing for equality, instantiating abstractions, comparing descriptions, and type-checking.

4.2.4.3 Uniformity

In Act2, similar things are done in similar ways. We have already seen the similarity of **new** and **a** expressions. Creating a new bank account with `(new bank-account (with balance 500))` is very similar to creating a description of the bank account with `(a bank-account (with balance 500))`

The **create**, **case-for**, and **one-of** expressions are quite similar in the way they choose one of many possibilities. They all have the form:

```
(introductory-part
 possibility-1-1
 possibility-1-2
 ...
 possibility-1-n1
 (otherwise possibility-2-1
            possibility-2-2
            ...
            possibility-2-n2
            (otherwise ...)))
```

The first set of possibilities, *possibility-1-i*, are tried concurrently. The first (temporally) to succeed is chosen, and its body of commands is evaluated. If none is successful, the second set of possibilities is tried. Any number of sets of possibilities

can be denoted in nested otherwise clauses. If none succeed, the evaluation complains.

Another aspect of uniformity is that **case-for**, **let**, **one-of**, and **if** expressions have exactly the same syntax and very similar semantics as **case-for**, **let**, **one-of**, and **if** commands. In addition, these expressions, and the **create** expression, have bodies of commands very much like those of composite commands. This allows concurrent activities to be performed as the bodies are evaluated. All bodies have the form:
do *command-1 command-2 ... command-n*

The idea of denoting the natural or exceptional "value" of a composite expression is thought of as sending a reply or complaint communication in response to a request for evaluation of the expression. Therefore, the same syntax is used for this as is used for replying or complaining in response to a request communication, in the bodies of **create** expressions. The **reply** and **complain** commands serve both purposes.

4.2.4.4 Programmer Productivity Supported by High-Level Constructs

Studies have suggested that the average amount of debugged code, measured in lines, a programmer can write per day is relatively constant across languages. The most interesting of these tests supported this result when comparing assembly coding and PL/1 coding. It found that people would write and debug at roughly the same rate in lines of code per day. Because a line of PL/1 code typically does much more than a line of assembly code, the PL/1 programmers tend to produce more. This might be attributable to the increase in readability, understandability, and programmability, as well as higher-level abstraction mechanisms.

Part of the goal in the design of actor languages is to do as much as possible

for a programmer. Writing highly concurrent programs in some languages, such as Mesa, assembly code, and even Ada, is somewhat cumbersome and requires special attention; concurrency is inherent in Act2 and needs little if any consideration by programmers. Act2's high-level constructs allow convenient expression of complex concurrent behavior.

Another of Act2's features is pattern-matching, which condenses and localizes much functionality in areas such as recognition, filtering, and dispatch. Act2 makes it potentially more of a savings, once assertion and deduction mechanisms are embedded and made use of.

4.2.4.5 Abstraction and Extension

Act2 has mechanisms for defining and instantiating abstractions, the **define** and **new** expressions. These mechanisms unify the notions of procedural, control, and data abstraction by emphasizing communication, rather than representation. Abstraction allows a programmer to define his own abstractions in addition to those which are provided with the language. Because of the uniformity in which pre-defined and user-defined abstractions are treated, this can be thought of as raising the level of the language itself. It makes the language more suitable for implementing applications which are more easily expressed in terms of those constructs.

Act2 goes beyond this aspect of expressiveness, allowing programmers to introduce new expressions and commands into the language itself. Not only is it possible to define abstractions suitable for special application domains, but it is possible to tailor the language itself into one allowing convenient expression of fundamental concepts in the application domains.

Programmers can exploit the extensibility mechanisms to provide a more comfortable language with syntactic sugaring allowing common behavior to be expressed concisely. We can extend the language with more specific constructs, which are implemented in terms of the more general ones. For example, the **if** construct is simply a specialization of **one-of**. It was, however, included because of the frequency with which binary decisions occur, and because it makes them more readable, and is more familiar to programmers.

4.3 Act2 has Actor Semantics

4.3.1 Act2 is Actor-Based

The Act2 language is based on a well-defined, mathematically understood computational model. The integrity and consistency of the actor model have been established in [Clinger 81b]. This formal model serves as a solid foundation for Act2, which inherits the benefits of well-definedness, and exploits the properties of the model.

Many of the fundamental issues in language design of a language system, such as abstraction mechanisms and concurrent computation, are dealt with abstractly by the Actor model of computation [Hewitt and Baker 78]. Because Act2 allows the characteristics of the model show through at the language level, issues handled by the model are inherited by the language. The language design can concentrate more on other issues. This is another of the features of a layered language design approach.

4.3.1.1 Representation Abstraction

An actor cannot directly view or manipulate the contents or implementation of another actor. All it can do is communicate with the actor, asking it for information or requesting it to change. Only the actor itself can alter its behavior. This property is known by several names, including representation abstraction, protection, encapsulation, opacity, and information-hiding. The hiding of implementation details has proven itself as one of the fundamental paradigms of software engineering.

Limiting access to an actor's implementation has many benefits in the area of software engineering. Techniques for data-type induction have been developed for the object-oriented computational model [Liskov 72, Guttag, Horowitz and Musser 76]. Similar techniques can be used within the actor model [Hewitt and Attardi 81, Hewitt, Attardi, Lieberman 79]. The correctness of an actor's implementation is a local phenomenon, depending only upon its specification, its script, and the specification of the actors it communicates with.

The discipline of communication enforced by actors allows the implementation of an actor to change, without affecting the actors which communicate with that actor, as long as the actor's communication protocols do not appear different to them. It also allows different implementations of an actor to coexist.

4.3.1.2 Absolute Containment

In addition to being opaque, an actor is entirely self-contained. It can only communicate with its acquaintances and with the acquaintances of the communication it is currently processing. There is no notion of global state to put restrictions on the existence and location of the actor. Actors can be migrated from

worker to worker when convenient, because of their machine independence. This transportability is possible precisely because there is no dependence of the actor on any storage locations local to a worker.

4.3.2 Modularity

Actors' properties of representation abstraction and absolute containment suggest the modularity inherent in the actor model. The model goes beyond this, unifying data, control, and procedural abstractions. The fact that an actor contains both data and procedural information (its acquaintances and script), is naturally sufficient for representing both procedures and data structures. The model's emphasis on communication blurs the distinction between them.

The emphasis on communication also allows the representation of control abstractions as actors [Hewitt 77]. One typical use for control structures in programming languages is to obtain a stream of values [Liskov, et al 81]. These can be represented as dynamic sequences in Act2, a literal manifestation of the "sequences" like those in [Waters 83]. Suppose we have an abstraction implementing tree traversal. We can simply create an actor representing the traversal of some specified tree. This actor might behave just like a sequence, accepting requests for its `first` and `rest`. In fact, it retains information about the tree and its placement within it, and computes the requested information dynamically.

Sponsors allow the implementation of a new class of control abstraction. They regulate the availability and rate of consumption of computing resources by asynchronous computations. Explicitly expressing this in the computations themselves would drastically increase the complexity of their implementations.

Act2 unifies the ideas of data, control, and procedural abstraction in a single abstraction mechanism. This abstraction mechanism encapsulates not only the creation of actors, but arbitrary expressions in the Act2 language. This allows for a more convenient expression of procedural abstractions than that mentioned above. The **define** and **new** expressions cooperate to provide this very flexible form of lambda abstraction.

Abstractions in Act2 are actors, and can be sent communications just like any other actor. This corresponds to the idea of abstractions being first-class objects in other languages. This is clear in the case that the abstraction definition simply represents the creation of an actor. It is also true in the case of some other arbitrary expression. For example, consider the definition of a `factorial` procedural abstraction as a recursive expression. The implementation is installed in a `factorial` atomic description, which can then be sent communications relevant to the implementation.

4.3.3 Message Passing Semantics Permeate Act2

In an actor-based language such as Act2, everything is an actor. All computation is performed using transmission of communications. These provide tremendous flexibility in expressing and performing computations, as will be discussed below.

4.3.3.1 Primitive Actors use Message Passing Semantics

In Act2, the message-passing paradigm of the actor model is used down to the level of primitive, pre-defined actors such as numbers and symbols. For example, simple arithmetic operations can be performed by the numbers themselves, in response to requests to do so. Because a uniform protocol is used throughout, a user

can define his own form of numbers, such as complex numbers, which behave like numbers. Code written for handling numbers in general will work even when some of the numbers handled are user-defined ones. The use of message-passing semantics in this manner makes the arithmetic operations work across machines, and with arbitrary actors using the numeric communication protocols. This is essential for concurrent applications in general. Arithmetic operations involving primitive numbers on a single worker is viewed as a special case which can be optimized, rather than as the only case, such as in many other languages.

4.3.3.2 Actors Implemented in Act2 have Actor Scripts

The script for an actor implemented in Act2 is itself an actor. The declaration of an abstraction involves the installation of an abstract syntax tree representing the abstracted expression. Instantiation of uncompiled abstractions causes this abstract syntax tree to evaluate itself. Actors created in this manner have scripts which are composed of a tree of abstract syntax actors, representing the behavior of the actor in terms of Act2 language constructs. Acceptance of a communication involves message-passing among the abstract syntax actors composing its script.

4.3.3.3 Programs as Data

Act2 programs are "first-class objects" in the Act2 language. User input is read in as symbols, numbers, and list structure. All of these are actors, which can be communicated with. Parsing produces abstract syntax trees, composed entirely of actors. Environments are first-class objects in the language, and can be accessed, created, or manipulated by programs. Evaluation can be done simply by sending an evaluation request to an abstract syntax tree. It is evident, then, that Act2 programs can be written which manipulate or create other Act2 programs. Such power accounts partially for the popularity of Lisp.

The **quote** expression is very useful for construction of Act2 code by other programs. It allows the denotation of unparsed list structure and symbols, of which Act2 syntax is composed. The **parse-expression** expression is convenient for denoting abstract syntax trees. It parses, but does not evaluate the list structure or symbols in its argument.

4.3.4 Transactions

All communication in Act2 occurs by one-way, asynchronous, buffered transmission of communications. It does not rely on a procedure-call mechanism, as do languages like Argus and Ada. Procedure call semantics can be implemented efficiently using message-passing. They are simply a special case of the more general notion of transactions in Act2.

Act2 supports three major kinds of communications. Request communications correspond roughly to the procedure-call part of the procedure--call-and-return mechanism. They include extra information, customers and complaint-departments, indicating where a response should be sent. Reply communications correspond roughly to the return part of the procedure-call-and-return mechanism.

A very common pattern of communication is the sending of a request, including a customer, to some target actor, followed eventually by the sending of a reply to the customer. The sending of the request and the sending of the reply are fully decoupled, however. The receiver of the request can redirect the request to another actor. It can do some processing and let another finish. It can hang on to or pass along the customer from the request, which is a "first class object" in the language. It, or some other actor, can eventually reply to the customer. Between the sending of the request and the sending of the reply, arbitrarily convoluted patterns

of communication transmission can occur. Actors are not arbitrarily restricted by strict control structures like procedure call and return.

4.3.4.1 Customer Chains versus Execution Stacks

There is no need for execution stacks in Act2. This functionality is subsumed by "chains" of customers — customers with customer acquaintances. When they receive a reply, they might eventually reply to their customer acquaintances. These chains are more flexible than execution stacks. Many such chains can exist. They can branch off into multiple customer chains. They can span workers. Portions of them can be migrated from worker to worker, independently from the rest. Being actors, they can be kept as acquaintances and communicated with.

The very common pattern of sending a request and accepting a reply are expressed very conveniently in Act2. The programmer does not need to explicitly construct customers for each request. Act2 expressions transform their procedure--call notation, and the contexts in which they occur, into the sending of requests with appropriate customers. This is done without programmer effort. Common patterns of communication among actors on the same worker can be optimized, increasing the efficiency of the transactions.

4.3.4.2 Complaint Handling

When an actor accepts a request, it is usually expected to respond. If processing of the communication completes without problems, a reply communication can be sent in response. If minor problems occur, it is often possible to reply with some meaningful message. If, however, irreconcilable problems do occur, some means is needed to indicate that fact, as well as to respond with some communication with a message which might indicate the reason for the

failure and provide any information which might be helpful to recover from the problem.

Act2 provides a special type of communication called a complaint communication to represent an exceptional response. This corresponds roughly to Clu signals, PL/1 conditions, or error codes. In keeping with the Actor model of computation, Act2 performs exception handling using the message-passing paradigm.

Act2 provides mechanisms for handling complaints. The primary one, the **case-for** construct, is for handling complaints generated by the evaluation of an expression, which we'll call the *guarded expression.* It recognizes complaints using pattern-matching. It performs an additional service by recognizing replies using pattern-matching. That is, the case-for construct makes use of the pattern-matching paradigm to recognize and extract information from responses to requests, whether they are replies or complaints. Along with this recognition is the selection of a body of commands to be evaluated once the response is obtained.

The **case-for** construct serves both as a dispatching mechanism for (replies to the evaluation of) the guarded expression and as a complaint-handling mechanism (if complaints are generated by the evaluation). In this sense, **case-for** unifies the notions of dispatching, complaint-handling, information extraction, and decision-making. For example, suppose we had a variation of the **account** abstraction defined in {section 2.3, page 34}, which included the new balance in deposit and withdrawal receipts. When making a withdrawal, we could use the **case-for** construct to handle a complaint or to take different actions based on the new balance:

```
(case-for (ask my-account (a withdrawal (with amount x)))
  (complaint (an overdraft) do ...)
  (is (a withdrawal-receipt
        (with new-balance (≡b such-that (< b 500)))) do ...)
  (otherwise
    (is (a withdrawal-receipt (with balance ≡b)) do ...)))
```

Act2 provides a mechanism for handling complaints from a command. This is very similar to a case-for command with has complaint handlers, exclusively. Rather than guarding an expression, this command guards another command.

Complaints are automatically relayed by constructs which do not explicitly handle them. In addition, this does not even cause a degradation in performance, because requests have both a customer and a complaint department. Replies are sent directly to the customer. Complaints are sent directly to the complaint department, with no need for winding down through a customer chain. This idea was suggested in [Lieberman 82].

Act2 may, itself generate complaints when this is appropriate and there is no convenient alternative. For example, if no handler is capable of accepting a communication, Act2 will complain to the communication's complaint department (if it is a request) or to the implementor.

4.3.5 Inherent Concurrency

The actor model, with its one-way, asynchronous, buffered model of communication, is inherently concurrent. The Act2 language preserves this inherent concurrency in its high-level constructs.

Whenever no ordering is necessary between the evaluations of separate commands and expressions, the Act2 definition does not impose one. This allows them to be evaluated concurrently, and their evaluations can proceed in parallel if

sufficient parallelism is available. The design of Act2 attempts to minimize dependencies among expressions and commands. Inherent concurrency is an important aspect of our actor language which distinguishes it from other modern programming languages, in which concurrency must be artificially generated, or requires special attention from the programmer.

4.3.5.1 Local versus Global State Change

As discussed above, change in Act2 is a local phenomenon. An actor can change its own behavior, but cannot directly manipulate any form of "global state". This permits more concurrency by reducing the necessary synchronization. Because change is local, the only synchronization necessary is for serializers to process one communication at a time. Allowing change to a global state would require additional synchronization among actors and transactions, to preserve the integrity of the global state.

4.3.5.2 Local Binding versus Assignment

Act2 has no assignment command. In addition, bindings established in an expression or command, such as **create**, **let**, and **case-for**, are not available outside that expression or command. Because of this, there are no timing constraints among distinct expressions and commands. These expressions and commands can be evaluated concurrently. An assignment command would introduce timing constraints among commands, requiring them to be evaluated sequentially.

4.3.5.3 Concurrent Commands and Shared Resources

When commands share a resource, such as a serializer, programmers may wish to rely on additional synchronization. For example, one command might cause

some actor to change its state, and the other might ask the same actor for some information. The programmer may wish the request for information to reach the actor after any communications sent to it by the first command have been processed. A programmer can impose an ordering upon commands using the **sequential** command. This should only be used when the programmer explicitly relies on such timing dependencies.

4.3.5.4 Concurrent Evaluation and Explicit Sequencing

Act2 is specified as an inherently concurrent language. For example, commands in a command body are evaluated concurrently. Sub-expressions in a command or expression are evaluated concurrently. In a set of pattern-matchers, such as in **let**, **case-for**, or **create** expressions, all evaluations of patterns and expressions and subsequent pattern-matching itself are done concurrently.

Create, **case-for**, **one-of**, and **if** also contain the **otherwise** clause as a convenient way to serialize sets of possibilities. [Theriault 82] had a similar mechanism, but used it as a mechanism for providing a default body. Act2 generalizes this into a full-fledged sequencing mechanism, from which providing a default is a trivial case. For example, it is easy both to provide a default, as in

```
(case-for x
  (is (a stack (with top ≡t)) do ...)
  ...
  (otherwise (is something do ...)))
```

and to prioritize the sets of possibilities, as in

```
(case-for x
  (is (a whole-number) do ...)
  (otherwise (is (an integer) do ...)
             (otherwise (is (a real) do ...))))
```

4.3.5.5 Resource Management

With the amount of concurrent activity produced by Act2, resource management is important. Act2 uses sponsors for resource management. Every communication contains a sponsor, which is charged for the processing of the communication. This requires cooperation from the underlying apiary architecture, which requires payment for processing each event.

Below is an example of Act2 code which explicitly deals with resource management. It is simply a reworking of the example in {section 3.2, page 46}. In this code, it is the sponsor from the evaluation request which pays for the evaluation of the contained expression, rather than the sponsor from the first communication sent to it.

```
(define (new DELAY-EXPRESSION (with expression ≡exp))
  (create-unserialized
    (is-communication
      (a request
         (with message (≡eval which-is (an expression-eval)))
         (with sponsor ≡s))
      do
      (reply (create
              (is-communication ≡c do
                (let ((≡value match
                        (using-sponsor s do
                          (reply (ask exp eval))))) do
                  (send-to value c)
                  (become value))))))))
```

4.4 Act2 Integrates Description and Action

4.4.1 Coexistence of Mechanisms for Description and Action

One important consideration in the design of Act2 is the unification of mechanisms for description with the imperative mechanisms of the actor model. Act2 integrates the fundamentals of Act1 and Omega, which are very different in

character. Act1 deals in an operational world of message-passing, actor creation, and behavior change. Omega deals with knowledge acquisition in a lattice of descriptions, and deduction based on installed relationships among them. It models change by creating more descriptions, but is incapable of actually implementing actors which can change. A language suitable for open systems, or concurrent applications in general, must combine both sets of ideas.

Act2 is built upon the actor model of computation. It has constructs for transmission of communications, for making simple decisions, for creating actors, and for self-replacement. Act2 also has actors which behave like atomic descriptions and instance descriptions, which it uses for their information containment properties, for their descriptive properties, and for a direct form of pattern-matching. Act2's abstraction mechanism, the **define** expression, establishes a relationship between the two worlds by associating a description with every actor, which corresponds to the actor's "type". Act2's pattern matching acknowledges this relationship, serving as a form of "type-checking" when appropriate.

For example, a bank-account actor created with the expression `(new bank-account (with balance 500))` might be described by the instance description `(a bank-account)` or by the instance description `(a bank-account (with balance 500))` if the implementor of the abstraction wished to allow the balance information to be revealed. The actor could be matched by a pattern of the form `(a bank-account)` in either case, and by a pattern of the form `(a bank-account (with balance ≡x))` in the second case, with the identifier `x` being bound to the balance, `500`.

4.4.1.1 Abstract Syntax for Description and Action

One of the problems in integrating the ideas from Act1 and Omega is a set of apparent name conflicts which arise in the constructs we desire Act2 to have. Note the relationship between the instance description `(a bank-account ...)` and the abstraction instantiation `(new bank-account ...)`. In the instance description, `bank-account` is some concept or atomic description. In the instantiation, `bank-account` refers to the implementation of bank-accounts, as previously declared in a **define** expression.

In addition, flexibility demands that we be able to have arbitrary expressions as the concepts of instance descriptions, evaluating to atomic or instance descriptions. It also demands that we have arbitrary expressions denoting the implementation of an instantiation. Because arbitrary expressions undoubtably include locally-bound symbols, `bank-account` is a symbol in both cases, and must be evaluated in the prevailing evaluation environment at that time.

The resolution of such conflicts is done in Act2 by interpreting `bank-account` as a symbol, and broadening our interpretation of atomic descriptions. The functionality of atomic descriptions is extended such that implementations are installed in them. This confirms the feeling that there is a relationship between the concept of `bank-account` and the implementation of bank accounts.

It does not prevent the coexistence of implementations of different bank accounts, which can be installed in different atomic descriptions. That is, I can have my own concept of bank-account, and a corresponding implementation, whereas you too can have your own concept of bank-account as well as your own implementation. Act2 will deal correctly with both of them.

Act2 could have introduced operators to denote atomic descriptions and

implementations. For example, `αbank-account` and `ιbank-account` might produce atomic descriptions and implementations by performing some calculation, perhaps looking them up in different environments. This would have reduced readability, and would have been a less general solution. As it stands now, users could extend the syntax of Act2 if such expressions were desirable.

4.4.2 The World of Action

Act2 contains constructs for transmitting communications. It has constructs for creating new actors with specified behaviors and acquaintances. The very nature of an actor is that of action. It receives a communication, then causes effects to happen. These effects might be communication transmissions or actor creations. Primitive actors provide a message-passing, actor-based interface to the underlying hardware. For example, some primitive actors might serve as an interface to a keyboard, a screen, or a robot arm. In response to communications, they might read characters, display information, or construct a ham sandwich.

4.4.2.1 Change

One important effect an actor can have is to cause itself to be replaced by another actor. This is such a significant concept that Act2 makes a fundamental distinction between serializers and unserialized actors. Serializers are Act2's method of dealing with change. They are treated very differently in most aspects of their existence, such as pattern-matching, equality tests, copying, migration, and concurrency.

4.4.2.2 Local Changes versus Global State Changes

As we have mentioned earlier, all change is local to an actor. This is done in

preference to using a state transition model, in which change happens to some global state. A serializer can replace itself with another actor, in response to some communication. We have seen the benefits of this for concurrency and software engineering.

4.4.2.3 Maintaining Computation Histories

Act2 provides a biography mechanism for recording the computation history of actors. This allows actors to keep track of the communications they receive and what they do in response, in a manner such that the effects of receiving the communication can be undone. This feature is an adaptation of the work reported in [Jefferson, Sowizral 82]. A serializer which has maintained its history can be rolled back to some previous state, or can report on previous states.

4.4.3 Descriptions as Information Containers

Part of the integration of Omega descriptions in Act2 is the use of instance descriptions as information containers. From this perspective, instance descriptions can be thought of as aggregate data types, or type constructors. They, like sequences, allow a programmer to express a collection of actors, without instantiating any special abstraction.

An instance description can be thought of as a flexible record structure. The concept serves as a tag indicating type. The attributes correspond to record fields, where attribute relations are field names, and where attribute fillers are field values.

In this capacity, instance descriptions are very convenient for packaging actors and representing information. They are especially useful as messages in communications. The use of instance descriptions for this purpose is especially

beneficial, in view of Act2's ubiquitous pattern matching. Even more important is the potential it allows. When inheritance and deductive mechanisms are embedded in Act2, it may be possible to have different perspectives on incoming information, or to coerce the incoming information into a usable form.

4.4.4 Description of Actors: Data-typing and Specification

4.4.4.1 Description of Actors

Part of the integration of Omega descriptions is the use of those descriptions to describe actors, much like types in common programming languages. This makes pattern-matching work nicely for both instance-descriptions and arbitrary actors. It also makes the idea of type potentially much more powerful and general than simply that of a tag, as it is in many existing languages. It opens an avenue for making assertions and deductions about the properties of a type and about the relationships among types, when inheritance and deduction mechanisms are added to Act2.

4.4.4.2 Behavioral Types

An actor's type is not simply a tag, but a description of the actor. Although Act2 itself does not implement assertion and deduction mechanisms, the perspective it takes on types is very important in the philosophy of the language, and provides tremendous potential for a very powerful and flexible type system. The notion of type in Act2 is of describing the behavior of an actor. An actor's type provides reliable information about the semantic properties of actors. With this philosophy and perspective, actual type mechanisms can range from simple descriptions and exact match, which looks much like a tag-oriented scheme, to behavioral

specifications ranging from partial to total descriptions of an actor's behavior. The full flexibility and generality of the description system can be brought to bear in the description of abstractions, including incremental and partial description and inheritance of properties from other descriptions. The type system can be arbitrarily finely-grained, and have a very flexible notion of type equality and type conformance. A form of pattern-matching can be used which treats the pattern match as a goal, then uses mechanisms for deduction in the description system for establishing or refuting that goal.

4.4.4.3 Controlling Visibility

By default, the description associated with an actor is simply a description with no attributes, such as `(a bank-account)` rather than a more explicit one such as `(a bank-account (with balance 500))` in order to preserve the opacity of the actor. Act2 provides alternative expressions for creating actors, so a programmer can allow the extra information to be revealed, to allow the extra information to be extracted in a pattern-match. Deductions can potentially be made which involve these attributes and fillers. One feature of this for serialized actors is that it allows convenient extraction of a consistent state.

4.4.5 The Many Uses of Pattern Matching

Pattern-matching plays a fundamental and pervasive role in Act2. It is used for extracting information from instance descriptions, for authentication, for type-checking and extraction of information from actors, for determining equality of actors, and for binding variables to actors. It is used for acceptance of communications, for dispatch on expressions, for exception-handling, and for decision-making. It is also used in the instantiation of abstractions, in the evaluation of **new** expressions.

The pattern-matching implemented by Act2 involves no deduction mechanisms, and is efficient when compared to deduction. Act2 makes it possible to embed deduction mechanisms in the language and implement an extended form of pattern-matching which makes use of them. This means that the full power of a description system like Omega can potentially be employed in the pattern-matching process.

4.4.5.1 Pattern-Directed Recognition and Extraction

In Act2, the standard mechanism for recognizing actors, which may or may not be instance descriptions, is to use pattern matching. Pattern matching works both on descriptions, using simple specialization axioms, and on arbitrary actors, using their associated descriptions. Dispatching on the characteristics of an actor can be done using pattern-matching. Recognizing communications can be done by pattern-matching.

4.4.5.2 Security

Security is an important part of modern programming languages. Type-checking allows detection of type errors, a very common nuisance in programming. Act2's abstraction mechanism allows the implementor to restrict what actual parameters are bound to what formals. Act2 allows the types of objects to be declared and performs type-checking wherever a programmer puts restrictions. Restrictions can be put anywhere a pattern or variable-binding can appear.

Act2 allows the programmer to refrain from making restrictions: to have objects described as `something`, and to make no restrictions or mild restrictions on bindings. It also allows a programmer to make very comprehensive restrictions on matching. They not only serve to allow more complete type-checking, but also

provide very good documentation for programmers reading the code. Presence of type information increases the potential for optimization.

4.4.5.3 Polymorphism

Abstractions are typically defined in terms of sub-abstractions. Polymorphism is that property of being able to use as a sub-abstraction any of a set of abstractions conforming to some expected behavior, rather than a single, pre-specified abstraction. This property is often thought of in terms of overloading, generics, and parameterized abstractions. Polymorphism is provided by Act2 by its use of message-passing semantics as its fundamental means of communication. This is similar to the way Smalltalk and Simula provide polymorphism. As long as an actor responds to all the right messages in all the right ways, it can be used. For example, an operation can be declared, which operates on some set of arguments. The operation can be performed successfully on any actors, as long as they behave in the correct manner.

4.4.5.4 Authentication

Act2 addresses the issue of authentication with atomic descriptions. Each creation of an atomic description, `(new concept (with name ...))`, creates a new atomic description which will not match any other atomic description. Everyone shares enough atomic descriptions so that they can communicate with each other, enough to mail atomic descriptions to each other.

Suppose you give someone access to a bank account you created — one implemented in {section 2.3, page 34}. Unless you give him access to the atomic descriptions used as the concepts in the communication-handler patterns (such as `balance`, `deposit`, and `withdrawal`), the bank account will not accept any

communication he sends to it. Remember that his atomic description with name `withdrawal` is not the same as the one with which you defined your bank-account abstraction. You can send him your `balance` atomic description, which he can bind to some identifier `his-balance`, and can then obtain your account's balance by sending it a request with message `(a his-balance)`. He will still, however, be unable to make withdrawals, because he does not have your `withdrawal` atomic description. Because of static scoping, you can define operations with access to certain bank account operations which a user of the operation does not have access to.

4.5 Act2 and Open Systems

4.5.1 Suitability for Open Systems

Prelude must have the full generality and flexibility of a language suitable for open systems [Hewitt, de Jong 82]. Act2, as a substrate for Prelude, must also be suitable for this style of programming, by realizing a suitable model of computation, providing sufficient fundamental mechanism, and providing a potential for embedding appropriate higher-level mechanisms and policies. A requirement for languages suitable for open systems is the ability for independent programmers to communicate with each other, selectively share independently-produced software and data, and merge subsystems together as integrated wholes. Open systems are characterized by the coexistence of independently-conceived and evolving software applications which need to cooperate in flexible but controlled ways. These applications might be autonomously owned by mutually suspicious organizations which never-the-less wish to share information and information processing abilities to some extent.

We believe Act2 is suitable for open systems. It provides a solid foundation upon which more sophisticated languages and applications can be implemented. It provides mechanisms for description as well as for causing effects and change. It also provides for a natural coupling of descriptions and actor systems. Its use of descriptions as a "type" mechanism means its fundamental perspective on types is quite flexible and potentially very powerful. With the addition of inheritance and deduction mechanisms, such a "type" can be extended to include extra semantic properties of the abstractions and relationships among abstractions. In the limit, a "type" might include a full specification of the abstraction. Similarly, this allows very flexible forms of identifier declaration, type checking, abstraction instantiation, and parameter passing. These features are extremely useful for programming-in--the-large. The potential flexibility of this type system would allow for the coupling of independently conceived and independently-named application systems, even in the presence of name conflicts.

Act2 supports controlled sharing and cooperation among independent systems. Part of the bottoming out of instance descriptions is a set of concepts which can be understood by all users of Act2. This enables independent programmers to communicate. Each programmer operates in his own environment, which is an actor itself. All environments are distinct. The opacity of actors insures that they cannot be compromised directly, without sending communications.

Authentication is a crucial part of controlled sharing among separate applications. Act2's atomic descriptions provide this important functionality. All atomic descriptions created by **defconcept** or **(new concept ...)** are distinct and do not match each other. This turns instance descriptions into key-based access mechanisms, in addition to the functionality they already have. Built into Act2's recognition facilities are mechanisms suitable for authentication of incoming communications. Because Act2 expressions are statically scoped, the environment

used to resolve names used in the concepts of patterns in an actor's implementation will be the environment in which the actor was defined. The actor will not be compromised simply by using it in another environment.

One of the most important requirements for languages suitable for open systems, and for Act2 in particular, is sufficient generality to express desired concurrent computations. By virtue of its actor foundation, Act2 inherits the properties shown about the actor model [Clinger 81b]. We have also demonstrated some aspects of its generality with initial experiments. For example, with our applicative language experiment, we showed that Act2 is more general than applicative languages for concurrent systems. We also implemented a shared checking account in Act2, which was suitable for concurrent systems could be shared among several owners.

This chapter has discussed a number of issues in the design of Act2. A few more issues are discussed in an appendix {section F, page 205}. All of these are germane to languages for open systems. The choices made in Act2 for dealing with these issues were all made with the goals of suitability for open concurrent systems, for expressibility of high-level applications, and for support of software engineering principles.

4.5.2 Synergy

It is interesting to note that many of the design decisions for Act2 had a bearing on several issues. Also, different design decisions had pleasant interactions which went beyond simple additivity. As a simple example, lexical scoping is not only more natural for programmers, but combines with our notion of uniqueness of atomic descriptions and pattern-matching to provide authentication. Our relationship between descriptions and arbitrary actors provides a very flexible type

system. It also makes pattern-matching suitable for type-checking, information extraction, identifier binding, accepting communications, catching exceptions, and dispatching on the values of expressions.

Chapter Five

Implementation Issues and Mechanisms

5.1 Bottoming Out

The previous chapter described issues which were explored in the design of Act2, assuming the actor model of computation as a foundation. The model is conceptually elegant and sound, and is inherently machine-independent. An actor language implementation, however, must bridge the gap between this conceptual model and a concrete computer architecture rooted in physical hardware. The actor model of computation, by its very nature, requires careful implementation for a practical realization. There are many potential circularities which must be unraveled so useful computation can take place. The following paragraphs will point out some of the circularities which might exist in a naive implementation attempt.

Actors interact by transmitting communications to each other. The communications themselves, as well as the messages they contain within them, are also actors. Our conceptual model dictates that actors cannot directly manipulate or read each others' contents. Potentially, an actor receiving a communication must send it (the incoming communication) further communications to find out what's in it, and the communication itself faces the same problem when it receives these communications. A good implementation must break this circularity in a general and flexible way.

In the conceptual, machine-independent actor model, every actor has a script, which is also an actor. Because it is an actor, the script itself must also have a script.

It is obvious that there must exist primitive actors at some level, but a proper solution must be sufficiently flexible to let programmers reference and manipulate scripts or define their own script actors.

Instance descriptions are Act2's primary information containment and recognition facility. To construct an instance description, we need an appropriate atomic description. To get the atomic description, we need to ask an environment for it, since all we have is its name. To ask the environment for it, we must construct an instance description for a message.

For a practical implementation, we represent numbers, symbols, and lists as themselves — as lisp objects. Such primitive actors must be able to behave like actors do, receiving requests and transmitting replies or complaints in response. These actors must be made to behave like Act2 numbers, symbols, and lists.

5.1.1 Rock-Bottom Actors

Much of the bottoming-out process must be done in intimate cooperation with the underlying computer architecture. The Apiary recognizes certain *rock-bottom actors*, such as numbers, lists, and symbols, whose concrete representation is inherited directly from the underlying implementation language, Lisp. These concrete representations do not contain scripts for processing incoming communications. When a worker is instructed to transmit a communication to such a primitive actor, the worker is responsible for handling the communication in some appropriate way. The worker itself can intercept and directly handle a few special requests intrinsically associated with these actors. The rest of the communications must be handled by Act2.

Part of the process of installing Act2 is the installation of actors to serve as

representatives for each kind of rock-bottom actor. When a worker cannot directly handle a communication for a rock-bottom actor, it asks a representative to handle the communication on behalf of the rock-bottom actor. The representative is responsible for realizing the expected behavior.

5.1.2 Scripts

The Apiary architecture supports the creation of actors with scripts composed of Lisp functions. These primitive scripts correspond to microcoded abstractions in many other programming languages. Many of the actors used in the implementation of Act2 have scripts which were written in Scripter, which compiles into Lisp.

Act2 deals with scripts in terms of abstract syntax trees — hierarchies of actors which represent Act2 code. Actors created by instantiation of uncompiled Act2 definitions have scripts composed of abstract syntax trees. These abstract syntax scripts must be integrated into the Apiary implementation, so these scripts can run on the Apiary. This is done with a special kind of primitive script with acquaintances including the abstract syntax script, the definition environment and a description of the actor. These actor-based scripts coexist with and are implemented using the primitive scripts.

5.1.3 Communications

Communications are primitive actors which are recognized and manipulated directly by the Apiary architecture. They serve simply as a package for transmission of information between actors, and are recognized as requests, replies or complaints. Communications contain special acquaintances, such as a message, a sponsor or a customer.

5.1.4 Instance and Atomic Descriptions

When an actor receives a communication, it typically attempts to recognize the communication's message by pattern-matching. The patterns used are quite often instance descriptions. An instance description serving as a pattern needs to ask the message some questions in order to discover whether it should match. Considering the common case where the message is an instance description, we must examine what might happen when an instance description receives a communication containing another instance description as a message. Does it naively exhibit the above behavior? Somewhere, we must break the circularity in which an instance description sends an instance description another instance description in order to find out what's in it. Much cooperation is needed from the Apiary to bottom out instance descriptions. In that respect, they resemble primitive actors like numbers and symbols.

Part of the cooperation we require from the Apiary is a primitive operation which determines, without causing any events, whether an actor's script is a special instance description script. If the message in an incoming communication is such a special instance description, its parts can be extracted directly by special Lisp ("micro-") code without causing more events. This special Lisp code can extract the instance description's concept as well as its attributes and their parts, using its knowledge of the the format in which instance descriptions are represented. Instance descriptions are represented as primitive, scriptless data structures easily recognized by Lisp code[1].

---

[1]This also means that these instance descriptions are very much like the primitive, rock-bottom actors. The Apiary must recognize primitive instance descriptions and make sure any communications transmitted to them are handled. Act2 must install representatives which handle communications directed to them on behalf of primitive instance descriptions.

Once the parts of the instance description have been extracted, the only barrier to understanding its meaning is the concept. Using atomic-description concepts exclusively would require causing at least one event to find out what it meant. In order to break the circularity we've just described, it is crucial that somewhere along the line *some* actor must be able to accept and recognize an incoming communication without causing any further events. Therefore, for some low-level instance descriptions, a symbol is used as the concept instead of an atomic description. All relevant information can be extracted from these easily-recognized "primitive" instance descriptions without causing any events. In an Act2 implementation, we write Lisp code for accepting communications. If it can extract all necessary information directly, it does so. If it cannot — because the concept of the message is an atomic description or because the message is not a directly accessible instance description — it actually resorts to a message-passing protocol for obtaining the necessary information.

Given that we have instance descriptions with concepts that can be either symbols or atomic descriptions, and that we want user-written code to be able to send messages to any actor, we need some uniform way to denote both kinds of instance descriptions. Moreover, we do not want to require users to distinguish between them. Once again, we have increased the functionality of atomic descriptions in order to solve yet another fundamental problem. This time, we add a piece of ("concept-creation") information which tells whether to create instance descriptions with a symbol or an atomic description for a concept. When an expression denoting an instance description is asked to evaluate itself and the sub-expression denoting the concept evaluates to an atomic description, it asks the atomic description for an appropriate concept. Based on its concept-creation information, the atomic description replies either with itself or with the symbol which is its name. By default, **defconcept** and **`(new concept ...)`** create atomic

descriptions which indicate that the atomic description should be used for concepts. Another expression, `(new primitive-concept ...)`, creates one which uses the name, a symbol, for concepts.

5.2 Extensibility from a Listen-Loop

The mechanism for extension of Act2 is the Act2 listener itself. This loop is implemented in an actor-based way, which naturally provides much flexibility. When the loop is started, an environment becomes associated with it. This is the environment which the listener uses to obtain keyword environments for parsing and to for resolving names when evaluating expressions. The listener first reads in a surface syntax representation — list structure, symbols, and numbers — which denote an Act2 expression. It then asks the surface syntax actor to parse itself into an abstract syntax actor representing the expression in a meaningful way. When this has been successfully completed, the listener asks the abstract syntax actor to evaluate itself as an expression, using the prevailing environment for resolving names to the intended actors. When the abstract syntax actor has replied or complained, the listener asks the response to print itself, then begins the next iteration.

The actor-based parsing technique is responsible for Act2's extensibility. When the listener asks a surface syntax actor to parse itself, it provides two environments to aid with the parsing. Symbols and numbers can ignore these environments, because they typically parse directly into themselves. Lists, on the other hand, represent more interesting expressions and commands. In order to increase the flexibility of parsing and to distribute the knowledge about different expressions and commands, lists solicit the help of more specialized parsers. The two parsing environments help realize this behavior by mapping symbols used as

keywords identifying a construct to an actor which can parse appropriate list structure into an abstract syntax actor representing the construct. One of these environments establishes expression keywords, and the other establishes keywords for commands. Syntactic extension of Act2 can be done simply by extending these environments appropriately. With this technique, lists can parse themselves simply by scanning themselves from left to right, looking for a keyword, then can ask the associated parser to take care of the rest of the parsing.

5.3 Providing both Positional and Keyword-Based Instantiation

The desirability of keyword-based and of positional notation for the instantiation of abstractions has already been discussed {section 4.2.3.4, page 61}. Act2 prefers the use of keyword-based notation, using the **new** expression. It also makes some provision for the use of positional notation, but discourages its use. For example, when `(factorial 3)` attempts to parse itself as an expression, it discovers that there are no expression keywords within it. As a result, it parses into an abstract syntax actor representing
`(ask factorial (an instantiate (with arguments [3])))`. Notice that this would handle multiple arguments nicely. The `factorial` atomic description will accept the `instantiate` request, then will proceed to match the arguments, in order, with the attribute fillers in the **new** expression template from the definition of factorial. From there, the instantiation will proceed as if the instantiation had been written as `(new factorial (with number 3))`, rather than as `(factorial 3)`, assuming the factorial had been defined as in
`(define (new factorial (with number ...)) ...)`. There is a risk in using positional notation in this way, however. If any subexpression should happen to look like an expression keyword, the expression would be sent to some undesired parser. For example, taking the factorial of the actor bound to the identifier **a**

would be written `(factorial a)`, and the list structure would be sent to a parser for instance descriptions[2]. Of course, it would be possible to extend the language with a new kind of expression with prefix notation which would have the right effect. For example, an expression of the form `(call factorial 3)` or `(do factorial 3)` could parse into abstract syntax for `(ask factorial (an instantiate (with arguments [3])))`. The leading keyword would eliminate the possibility of confusion.

5.4 Making Composite Constructs Work

One-of, **case-for**, **let**, **if**, and **using-sponsor** are examples of Act2 constructs which can be used both as commands and as expressions. They provide much flexibility and convenience in the language. We have already seen the **if** construct used as an expression in an implementation of a `factorial` actor {section 2.1, page 31}, and as a command in an implementation of an `account` actor {section 2.3, page 34}. If necessary, please refer to {section C.9, page 148} for a description of these constructs.

The commands which are allowed in the bodies of these constructs, and the interpretation of some of those commands, depend upon the context in which the constructs appear. Our implementation must resolve these problems and must enforce additional restrictions required by each context. For example, we never want to evaluate more than one **become** command, because that is semantically incorrect (not to mention confusing). If we encounter no **become** command in a serializer's communication handler, we must be aware of when processing has

---

[2]Fortunately, Act2 would have warned the programmer about a previous attempt to bind **a**, because it is used as an expression keyword.

finished, so we can prepare the serializer for accepting another communication. No **become** command should ever appear in an expression body context or in a communication handler for an unserialized actor. For constructs used as expressions, we want to evaluate exactly one **reply** or **complain** command, which represents the response from the evaluation of the expression. In the context of a communication handler, **reply** and **complain** commands transmit communications to the customer or complaint department from an incoming request. All of these problems must be taken care of in a suitable implementation of these constructs, even in the face of syntactic and semantic extensions by users.

The way we deal with these problems in our Act2 implementation[3] is to introduce a notion of *effects*, which represent the evaluation of commands. Whenever a command receives an evaluation message, it evaluates itself to the best of its ability, then replies with an effect representing what has been done and what remains to be done. An effect can be a simple instance description containing some information by convention, or can be a sequence of effects. For example, commands such as **reply-to**, which have enough information to completely evaluate themselves, reply with `(a completed-command-effect)`. Because it does not contain target information, a **reply** command such as `(reply` *message*`)` evaluates *message*, wraps it in a reply communication, then replies to the evaluation request with `(a send-effect (with communication ...))`. A **become** command such as `(become` *exp*`)` evaluates *exp*, then replies to the evaluation request with `(a become-effect (with replacement ...))`. A composite command replies to an evaluation request with the sequence of effects resulting from the body of commands it evaluates.

---

[3]Details can be found in the meta-circular description in {section D, page 163}.

The other portion of this solution is the processing of effects. There are two contexts in which commands may be evaluated — in expression bodies and in communication handler bodies. When composite constructs are evaluated as expressions, they select a body to evaluate, ask the commands to evaluate themselves, and collect the resulting effects in a sequence. It then recursively processes this effect, ignoring completed-command-effects, complaining if a become-effect is seen or unless exactly one send-effect is seen. If a single send-effect is left at the end of this process, the communication it contains is sent to the customer or complaint-department of the evaluation request, as appropriate. When an actor accepts a communication, the body of the chosen communication handler is evaluated, and the effects are collected in a sequence. It then recursively processes this effect, ignoring completed-command-effects, transmitting the communications in send-effects to the customer or complaint department if the incoming communication is a request (or complaining if it isn't), and complaining if more than one become-effect is seen. Afterward, if a become-effect has been encountered, the (serialized) actor can change its state.

5.5 Serialized and Unserialized Actors

Serialized and unserialized actors are actually represented and manipulated differently by the Apiary architecture. For example, no synchronization is necessary for unserialized actors, so they can process different communications concurrently, and can be copied indiscriminately. Different copies of an unserialized actor can exist on different workers, and are recognized as "the same actor," for purposes of matching and equality tests. Serialized actors, on the other hand, need synchronization. They can only process one communication at a time. Because they can change, serializers cannot be copied arbitrarily.

When a communication is sent to a serializer, it is enqueued on the serialized actor's incoming communication queue for processing. If the actor is not processing another communication, it will begin processing this one. When it has finished — and it is necessary for it to know when it has finished — it must prepare for processing the next communication. If it must become another actor, it forwards any communication in its queue to that actor. Otherwise, it begins processing the next communication in the queue. If the queue is empty, the serializer goes into a dormant but receptive state until it receives another communication.

5.6 Missing Information

Some Act2 expressions and commands do not require a programmer to explicitly denote all of the information needed for complete evaluation of the construct. The missing information is instead obtained implicitly from the context in which the evaluation occurs. For example, any construct for sending a message, such as **reply-to**, **complain-to**, or **ask**, must provide a sponsor to pay for the communication and its processing.

A **new** expression inside a **become** command need not fully specify the new actor, if it is of the same type as the current actor. Suppose we define a shared bank account actor with a **define** expression of the form,

```
(define (new checking-account
             (with balance ≡b)
             (with owner ≡o))
  ...)
```

If we have a **become** command like

`(become (new checking-account (with balance x)))` which does not mention the `owner` acquaintance, Act2 will create the new checking account actor with the same owner. What we wrote is assumed to mean

```
(become (new checking-account
            (with balance x)
            (with owner o)))
```

The missing information is filled in from the properties of the current actor.

In both cases, the way the information is provided to the commands is similar. An actor receives a communication, and chooses a body of commands to evaluate {section D.10.4, page 192}. The evaluation message it sends to each of the commands contains not only an evaluation environment, but also includes sponsorship information and a description of the instantiation with which the actor was created {section D.10.7, page 196}. This extra information is available to any command which cares to look at it. Implementations of abstract syntax actors are expected to relay this information. The way the **become** command transmits extra information to the **new** expression is to include it in the expression-eval message it sends to the expression it contains. Once again, abstract syntax actors for expressions are expected to relay any extra information in eval messages.

Another form of missing information in Act2 is the provision of default communication-handlers for standard protocols required of all actors. For example, all user-defined actors will respond to requests with messages such as `(a match ...)`, whether or not the implementor explicitly provides one. A programmer can explicitly handle any of these messages by including her own request handler (which looks just like any other request handler) for it in the first set of handlers in a **create** expression. Otherwise, Act2 will handle them immediately after looking in the first set of handlers for one which can accept the communication. These default handlers are built into the evaluation of create expressions. For more details, see {section D.10.4, page 190} and {section D.10.4, page 191}.

Yet another transfer of extra context information occurs on installation and

instantiation of abstractions. This is the mechanism by which "types" are associated with newly-created actors, even though the **create** expression itself has no such information. When a **define** expression, such as
`(define (new account (with balance ≡b))` *exp*) is evaluated, the abstract syntax for the new expression is asked to install the abstracted expression with the prevailing environment {section D.4, page 168}. This asks the **account** atomic description to install, among other things, the instance description `(an account (with balance ≡b))`. When an expression such as `(new account (with balance 50))` is asked to evaluate itself, it retrieves the installed implementation information {section D.4, page 169}. It creates an instance-description, `(an account (with balance 50))`, which it matches against the retrieved description, `(an account (with balance ≡b))`, then extends the definition environment with binding of `b` to `50`. Finally, it asks the abstracted expression to evaluate itself in the extended environment, with the description `(an account (with balance 50))`. When asked to evaluate itself, a create expression will use this description information if it is present; otherwise, it will use `something` as its description.

5.7 Actors and Types

When an actor is created as a result of the evaluation of a **new** expression, the **create** expression which actually creates the actor is provided with a description of the instantiation, such as `(an account (with balance 50))`. It remembers all or part of this description as a descriptor for pattern-matching. This corresponds to data types in some programming languages. By default, it would only use `(an account)` as a descriptor, to preserve the opacity of the actors. A programmer may also wish to make the attributes visible. Variations of the **create** expression — the **create-visible** and **create-visible-unserialized** expressions — do this, using the

whole instantiation description as a descriptor for the newly-created actors. The extra attribute information is then available for extraction in pattern-matching.

5.8 Making Pattern-Matching Work

Although basic pattern-matching in Act2 involves no deduction, it does cope with both simple matching of instance descriptions and matching which corresponds to type-checking. By "simple" matching of instance descriptions, we mean that the concepts match without deduction, that all attributes in the pattern are present in the object, and that corresponding fillers match "simply". An instance-description used as a pattern will match any instance description which is a simple specialization of it, and will match any actor which it describes (any actor whose descriptor matches).

Whenever pattern-matching occurs, there is always the possibility that part of the matching process will involve binding symbols to actors. The pattern-matching process involves a *pattern*, an *object* to match, and an unserialized environment layer which is extended with symbol/actor bindings as the match proceeds. Pattern-matching happens as follows: the pattern is sent
`(a match (with object O) (with bindings B))`, where `B` is `(new empty-layer)`. What happens from there depends on the behavior of the pattern.

If the pattern, `P`, is an instance description, the effect of the match will depend upon whether or not the object, `O`, is an instance description. If it is an instance description, we would simply want to compare them; otherwise, we wish to perform a "type check" of `O`. The pattern has no way of knowing what to do, so it has the object do the work by sending it
`(a converse-match (with pattern P) (with bindings B))`. If `O` is an instance

description, it will do a simple instance description match. Otherwise, the default handler provided by Act2 for `converse-match` will simply relay the `converse-match` request to the actor's descriptor, which is an instance description and will do a simple instance description match.

When an instance description, `O`, receives a converse match with bindings `B` and with pattern `P`, it asks the pattern for its concept, which it asks to match `O`'s concept with bindings `B`. Next it asks `P` for a sequence of attribute relations. For each relation: if `O` has no attribute with that relation, the match fails. If it has one, it asks `P` for the corresponding filler pattern, which it asks to match the filler of its own corresponding attribute. The bindings resulting from each match are fed into the next, until the matching is finished. If any of the sub-matches fails, the whole match fails; otherwise, the match succeeds with the bindings established during the filler matches.

5.9 Compilation

Act2 views Compilation as an unobtrusive optimization technique. A user should see no functional difference before and after compilation, even when debugging code. Compiled code should retain enough information about the source code from which it was generated to provide intelligible interaction with the user, who only deals in terms of the source code. Compilation is done in terms of abstractions. For example, an abstraction defined in a manner such as `(define (new account ...) ...)` is compiled simply by saying `(ask account (a compile))`. The atomic description then asks its installed implementation to perform the appropriate transformations.

Compilation could be done with arbitrary sophistication. Even the simplest

forms of compilation can provide substantial performance improvement. One optimization is to eliminate the transmission of as many evaluation and match messages as possible by inline expansion of the code from abstract syntax objects. Another is to switch from deep binding of free names in the definition environment, to straight indexing of acquaintances. Performing these two transformations should provide a large return on investment by substantially reducing the amount of message-passing that goes on. More optimizations can be added for economizing on events, and more conventional optimization techniques can be brought into play.

5.10 The Ubiquitous Atomic Description

It is worth a brief enumeration of the functionality of atomic descriptions, to gain a better perspective of just how useful they are in our implementation of Act2. First of all, of course, they behave like atomic descriptions, representing some abstract concept or individual in some user's model of a world. Because of this, they serve as suitable concepts for instance descriptions. Atomic descriptions are used as part of Act2's machinery for establishment of abstractions. Definition of **new** expressions involves installing implementation information in an atomic description. This contributes to a smooth coexistence of **new** expressions and instance descriptions. Atomic descriptions contain information pertaining to the creation of instance descriptions, contributing to the solution of the bottoming out problem for instance descriptions. In addition, the uniqueness and opaqueness properties of atomic descriptions combined with their use as instance description concepts provides an authentication mechanism for controlled sharing of actors in open systems. Compilation is done in terms of atomic descriptions. To improve the performance of a user-defined abstraction declared as
`(define (new foo ...) ...)` we simply ask the instance description, `foo`, `(a compile)`. Atomic descriptions help make clear the relationship between actors

and descriptions of those actors. They also provide a tremendous organizational function for an implementation of the language.

Chapter Six

Conclusion

6.1 Summary

Act2 is a highly concurrent programming language designed to exploit the processing power available from parallel computer architectures. The language supports advanced concepts in software engineering, providing high-level constructs suitable for implementing artificially-intelligent applications. Act2 is based on the Actor model of computation, consisting of virtual computational agents which communicate by message-passing. Act2 serves as a framework in which to integrate an actor language, a description and reasoning system, and a problem-solving and resource management system.

We have completed a design of Act2 and have implemented a preliminary version of an Act2 interpreter. The development process was interesting in its own right. In the absence of Scripter, the language we eventually used for implementing Act2, we were forced to complete the design without experimenting with an implementation. Instead, we created and evolved a meta-circular description of Act2 — an implementation of Act2 in itself. This served as our primary design tool, as our informal language specification, as our design documentation, as exploratory Act2 code, and as a medium with which to explore implementation strategies for the language. This was followed by an implementation in Scripter of a variation of a subset of the Act2 language, which served as a minimal-inertia test bed for ideas before a full-scale implementation was begun.

Act2 was designed to address basic actor language issues, and to be

syntactically and semantically extensible. Because of this, it can serve as a substrate for embedding more sophisticated language features — in essence allowing language designers to tailor their own languages, concentrating on the issues and mechanisms they care about and taking for granted the more fundamental issues which Act2 has already addressed. Because of these open-ended requirements, generality and flexibility were considered the most important issues in the design and implementation of the language. Act2 must have sufficient expressive power to implement as broad a range of actor systems as possible, and must be sufficiently flexible to permit (and encourage) sophisticated and fundamental extensions as yet unanticipated.

Act2 is based strictly on actor semantics. As a result, the language can exploit its well-defined, formally specified foundation. One obvious advantage is the unification of procedural, data, and control abstractions. Another is the inherent machine-independence and concurrency of the model, as well as the tremendous flexibility of asynchronous, unidirectional, buffered communication primitives. The permeation of the actor model down to the fundamentals of the language itself give it much generality. The emphasis on communications makes the language suitable for implementing open application systems.

Act2 integrates mechanisms for description with mechanisms for causing action and change. It uses descriptions for their information-containing ability, as well as for describing actors. The result is a very powerful and flexible notion of "type", and use of the pattern-matching paradigm to provide an extensive range of functionality for the language. Pattern-matching is used for recognition of actors, extraction of information from actors, binding identifiers to actors, accepting communications, authentication, catching complaints, dispatching on the values of expressions, type-checking, parameter-passing, and comparing actors.

Act2 decouples syntactic issues from semantic issues. This helps isolate the related sets of issues, so they can be addressed separately. Our approach immediately provides several advantages. Extension of the language is natural and easy, consisting simply of extending an appropriate environment with a binding of a new keyword symbol to a parser for the expression. It is possible to have alternative syntaxes for the language, and to have them coexist. It is more convenient to develop presentation and editing tools which work on abstract syntax, providing different perspectives on and different concrete manifestations of the abstract syntax.

Act2 supports modern software engineering principles. An interpreted, interactively oriented language, it encourages a conversational style of programming. English-like, but structured, syntax increases readability. A single abstraction mechanism (the **define** expression) unifies the ideas of procedural, data, and control abstraction. The mechanism for instantiation of abstractions (the **new** expression) labels each parameter. Act2 allows programmers to declare and check the types of actors denoted by identifiers or expressions. A very flexible notion of "type" ranges from no type at all to a full specification.

6.2 Design Philosophy

There was a definite perspective from which we approached issues, problems, and proposed solutions during the design of Act2. There was always a deep concern for generality and flexibility. There was a concern for programmability and economy of mechanism. This raised such questions as:

Can this feature allow more to be done?
Can parts of these mechanisms be replaced by user-defined actors?
Can these mechanisms be combined easily to provide new functionality?
Can this functionality be achieved with existing mechanisms?
Can this be done more easily and naturally?
What new mechanisms need introduction?
How do they interact with other mechanisms?
Do they address other issues?
Can we now remove or hone down some other mechanism?
Is this construct natural to use?
Does it do enough?

It was often necessary to postpone major decisions until others had been made. Similarly, it was sometimes necessary to reconsider previously made decisions in light of new ones. Often what initially seemed like a sticky problem, when left alone for a while, would eventually be at least partially solved by solutions to other problems. In fact, the synergy of concepts and mechanisms in the final product is a testimonial to the power and applicability of the underlying ideas, which were previously developed by the Message Passing Semantics group.

6.3 Future Work

There are still many problems which need to be resolved in the design and implementation of Act2. We'll pick an obvious example — attribute relations are currently used directly, without evaluation. It is clear that at least some evaluation will be necessary in the future, to allow attribute relations such as:

```
(with (owner of possession) fred)
(with (new ...) ≡x)
(with v ≡x)  ;; where "v" is an identifier.
```

One solution is to use identifiers bound to atomic descriptions for relation names, as we did for instance description concepts. Unfortunately, this has its own drawbacks. It requires more concepts to be defined. It restricts the programmer's choice of

identifier names, because name conflicts would occur if local variables had the same names as attribute relation names.

Biographies have only been partially implemented. Also, work needs to be done on deciding what kinds of history-oriented services should be provided by default for actors. We need to implement compilation. Some preliminary work has been done on this, but much more needs to be done about compilation and optimization before large scale implementations can be written and run in Act2. Work is needed on a programming environment and user interface to Act2. We are in need of source-level debugging tools for Act2, which are capable of dealing with concurrent activity spanning across workers. Interested readers are encouraged to approach the Message Passing Semantics group with suggestions.

Appendix A

Glossary

acquaintances The set of actors accessible by an actor. See {section 1.1.1, page 16}.

Act1 A computer language for expressing basic actor-based computations, implemented by Henry Lieberman. See {section 1.3, page 19}.

Act2 An extensible actor language which integrates basic mechanisms from Act1, Omega, and Ether. The Act2 programming language and interpreter are the main topics in this document. See {section 1.7, page 29}.

actor A virtual computational agent, which is machine-independent and communicates using message passing semantics. See {section 1.1.1, page 15}.

apiary A computer architecture consisting of some number of workers (processors) interconnected by high-bandwidth links. It is responsible for storage management and communication transmission. See {section 1.6, page 26}.

atomic description A concrete actor which represents an abstract concept or individual, for purposes of knowledge representation as in Omega. For example, an atomic description with name `automobile` can represent the abstract concept of *automobile*. It can be used as a concept in instance descriptions such as `(an automobile (with color red))`. See {section 1.4, page 21}.

attribute A part of an instance description, which specializes the description. For example, `(with color red)` is an attribute in the instance description, `(an automobile (with color red))`. See {section 1.4, page 22}.

attribute kind Part of an attribute which indicates the significance of the attribute to the instance description. There are different kinds of attributes, and different axioms can be applied to them. For example, `with` is an attribute kind in `(with color red)`.

attribute relation Part of an attribute which indicates the relationship of the attribute filler to the instance description. For example, `color` is an attribute relation in `(with color red)`.

attribute filler Part of an attribute which denotes a description or actor which is related in some manner to actors described by the instance description. For example, `red` is an attribute filler in `(with color red)`.

behavior A characterization of an actor, denoting what communications it can accept and how it will process each of them. See {section 1.1.1, page 16}.

binder An Act2 expression of the form `(bind` *symbol*`)`, which is used in pattern-matching to bind a symbol to the corresponding component of the object being matched.

biography A record of the history of an actor. This includes the communications the actor accepted, and the effects it caused in processing each of them.

communication A unit of information flow between actors. A communication is an actor containing information for another actor. It is transmitted from one actor to another as part of a computation. See {section 1.1.1, page 16}.

complaint A type of communication used by convention in an actor language to indicate that the processing of a request has not been successfully completed, and why. See {section 1.1.2, page 17}.

complaint department
An actor included in a request communication, which will accept complaints generated during the processing of the request, and react appropriately, continuing the computation of which the request was a part. See {section 1.1.2, page 17}.

concept In reference to instance descriptions, the abstract idea or concept of which actors described by the instance description are specializations. For example, `automobile` is the concept of the instance description, `(an automobile (with color red))`. See {section 1.4, page 21}.

customer An actor included in a request communication, which will accept replies from the processing of the request, and will continue the computation of which the request was a part. See {section 1.1.2, page 17}.

description A representation of some abstract concept, individual, or collection of individuals with specified properties. See {section 1.4, page 21}, and the definitions of *atomic description* and *instance description.*

dissemination The transmission of goals and hypotheses to interested sprites. See {section 1.5, page 23}.

Ether A reasoning system for concurrent systems, implemented by William Kornfeld. The reasoning process is modeled after the problem-solving activities typical of scientific communities. See {section 1.5, page 23}.

event The acceptance of a communication by an actor for processing. See {section 1.1.2, page 18}.

generalization In a description system such as Omega, a generalization of a description is another description which describes at least those individuals described by the first. For example, the statement that `((an automobile) is (a moving-object))` establishes `(a moving-object)` as a generalization of `(an automobile)`. It automatically relates the knowledge we have about `(a moving-object)`, such as how it obeys physical laws of motion, to `(an automobile)`. See {section 1.4, page 22}.

goal In Ether reasoning, a characterization of some problem to be solved, or some statement to be proven. See {section 1.5, page 23}.

history A record of the events in an actor's "life". See *biography*.

hypothesis In Ether reasoning, a characterization of something which is thought to be true. See {section 1.5, page 23}.

inheritance In reference to Omega knowledge representation, if an inheritance relation is asserted between one description and another, then all individuals described by the first are also described by the second. Any information collected about the second also applies to the first. See {section 1.4, page 22}.

instance description A representation of some set of related abstract individuals. Sometimes used to represent an arbitrary member of that set. For example, the following might be used to represent the set of red automobiles or any arbitrary red automobile, depending on the context of usage: `(an automobile (with color red))`. See {section 1.4, page 21}.

matching See *pattern matching*.

message Part of a communication. The message is that piece of information intended to be interpreted by the target of the communication. See {section 1.1.2, page 17}.

Omega A description system for knowledge representation and manipulation, implemented by Gerald Barber. It allows assertions to be made about the relationships between abstract concepts and individuals, and is able to make its own deductions based on these inheritance relationships. See {section 1.4, page 21}.

pattern See *pattern matching*.

pattern matching The process of determining whether some *pattern* is a generalization of some *object*. The pattern and object of the match may be a description or any other actor.

Planner An early programming language for Artificial Intelligence applications. It provided mechanisms for reasoning, but relied

on back-tracking to simulate non-determinism.

Plasma The first programming language based on the actor model of computation. See {section 1.2, page 19}.

Prelude An actor language which will have the full functionality of Act1, Omega, and Ether. It will be implemented as a set of extensions of Act2. See {section 1.7, page 28}.

reply A type of communication, used by convention as a response to a request which has been successfully completed. See {section 1.1.2, page 17}.

request A type of communication used by convention to initiate a form of communication resembling two-way communication. An actor sends a request communication to some target, expecting some response as part of the processing. The request contains a customer and complaint department which will accept the response and continue with the rest of the computation. See {section 1.1.2, page 17}.

response A *reply* sent to a *customer*, or *complaint* sent to a *complaint department*, as part of the processing of a *request*. See {section 1.1.2, page 17}.

script That portion of an actor which dictates what communications the actor can accept and how it will process each. See {section 1.1.1, page 16}.

serialized actor An actor which can replace itself with another actor, as part of the processing of some communication. See {section 1.1.1, page 16}.

serializer A serialized actor. See {section 1.1.1, page 16}.

specialization A description is a specialization of another description if the set of abstract or concrete individuals it describes is a strict subset of that described by the other description. For example, the statement that `((an automobile) is (a moving-object))` establishes `(an automobile)` as a specialization of `(a moving-object)`. It automatically relates the knowledge we

have about `(a moving-object)`, such as how it obeys physical laws of motion, to `(an automobile)`. See {section 1.4, page 22}.

sponsor A resource management agent. The apiary charges for each event processed. Every communication transmitted contains a sponsor, to pay for the processing of the event. See {section 1.5, page 24}.

sprite An independent problem-solving agent, which actively applies a specific problem-solving rule. Each sprite has a trigger pattern characterizing which goals or hypotheses which activate it, and a body indicating what to do if such an announcement is disseminated. See {section 1.5, page 23}.

target In reference to transmission of communications, the target is the intended recipient of a communication. See {section 1.1.2, page 17}.

transactions 1. Common patterns of communication, such as request/reply and request/complaint.
2. A computation. All activity caused by the sending of a request. See {section 1.1.2, page 17}.

unserialized actor An actor which cannot replace itself with another, in response to some communication. See {section 1.1.1, page 16}.

worker An independent processor in an apiary architecture. See {section 1.6, page 26}.

Appendix B

A Sample Session with Act2

A person interacts with Act2 by conversing with an Act2 listen loop. When the listener prompts for input, the person simply types in any Act2 expression. This expression is evaluated by the listener, which is associated with an environment serving as a context for resolving names. The listener displays the response it received from the evaluation, then prompts for more input.

In this appendix, we will present a "sample session" of conversational interaction with an Act2 listener. This will be slanted in order to gradually introduce the constructs in the language. Each iteration will have user input labeled *request* and Act2's response labeled *reply* or *complaint*. Brief commentary may also be interspersed with iterations, to reveal their significance or explain what's being input.

Like most languages, Act2 has numbers. Primitive numbers can be denoted directly, by a sequence of digits, optionally preceded by a minus (-) sign.

Request:
```
10
```
Reply:
```
10
```

Arithmetic operations are defined for numbers, and convenient expressions are defined for denoting them. Although both prefix notation, as in `(+ 10 7)`, and infix notation, as in `(10 + 7)`, may be used, prefix notation is recommended.

Request:
```
(+ 10 7)
```
Reply:
```
17
```

One can also directly ask the numbers to perform the operations. In fact, that is exactly what expressions like `(+ 10 7)` do when evaluated.

Request:
```
(ask 10 (a + (with operand 7)))
```
Reply:
```
17
```

The identifier, `true`, is bound to a logical value representing truth.

Request:
```
true
```
Reply:
```
T
```

The identifier, `false`, is bound to a logical value representing falsity.

Request:
```
false
```
Reply:
```
NIL
```

Symbols can be denoted by quoting them to prevent their evaluation. Capitalization may be used arbitrarily, because case is ignored when distinguishing symbols.

Request:
```
'x
```
Reply:
```
X
```

Act2 has sequences for representing ordered collections of objects. When an expression denoting a sequence is evaluated, each of the expressions denoting an element of the sequence is evaluated. By default, sequences are represented directly as lists.

Request:
```
[4 true false (+ 1 2)]
```
Reply:
```
(4 T NIL 3)
```

A simple expression exists for binding a symbol to some value in the listener's

top level environment. Only the expression denoting the value is evaluated.

Request:
```
(defname x (+ 7 5))
```
Reply:
```
X
```

Names previously bound in the listener's environment may be mentioned in expressions for the listener to evaluate.

Request:
```
(+ 5 x)
```
Reply:
```
17
```

The evaluation of some expressions may result in a complaint instead of a reply. The expression below attempts to divide by zero, which is not mathematically defined.

Request:
```
(// 5 0)
```
Complaint:
```
(a division-by-zero)
```

The **let** construct provides a convenient means for binding symbols to actors, for use within a body of commands. It is quite flexible, allowing an arbitrary pattern-match instead of just a trivial symbol binding.

Request:
```
(let ((≡x match 3)
      ((a foo (with bar ≡y)) match (a foo (with bar 4))))
  do (reply (+ x y)))
```
Reply:
```
7
```

The **defconcept** expression below binds a new atomic description with name `add5` to the symbol, **add5**.

Request:
```
(defconcept add5)
```
Reply:
```
ADD5
```

Now, we can define a new abstraction which adds 5 to a number we provide.

Request:
```
(define (new add5 (with number ≡n))  (+ 5 n))
```
Reply:
```
ADD5
```

We can instantiate our new abstraction with a **new** expression.

Request:
```
(new add5 (with number 2))
```
Reply:
```
7
```

An attempt to instantiate our abstraction with a form of the **new** expression which does not match will result in a complaint.

Request:
```
(new add5 (with bar 'x))
```
Complaint:
```
(a failure ...)
```

We can also define a bank account abstraction, as in {section 2.3, page 34}. We will assume all appropriate atomic descriptions already exist.

Request:
```
(define (new account
             (with balance ≡b))
  (create
   (is-request (a balance) do (reply (a balance)))
   (is-request (a deposit (with amount ≡a)) do
     (become (new account (with balance (+ b a))))
     (reply (a deposit-receipt (with amount a))))
   (is-request (a withdrawal (with amount ≡a)) do
     (let ((≡new-balance match (- b a))) do
       (if (≥ new-balance 0)
         (then do
           (become (new account (with balance new-balance)))
           (reply (a withdrawal-receipt (with amount a))))
         (else do
           (complain (an overdraft))))))))
```

Reply:
```
account
```

Now, we can create a new account, binding it to a symbol in the top level

environment for later reference.

Request:
```
(defname my-account (new account (with balance 30)))
```
Reply:
```
MY-ACCOUNT
```

We can deposit some money in our account.

Request:
```
(ask my-account (a deposit (with amount 5)))
```
Reply:
```
(a deposit-receipt (with amount 5))
```

We can also withdraw money from our account.

Request:
```
(ask my-account (a withdrawal (with amount 10)))
```
Reply:
```
(a withdrawal-receipt (with amount 10))
```

If we try to withdraw too much money, our account will complain.

Request:
```
(ask my-account (a withdrawal (with amount 100)))
```
Complaint:
```
(an overdraft)
```

Finally, let's find out what our current balance is.

Request:
```
(ask my-account (a balance))
```
Reply:
```
25
```

Appendix C

Act2 Language Description

This section informally presents the meaning of Act2 constructs in English. Precision is sacrificed for readability. As a result, parts of this informal description may seem ambiguous to some readers, who are invited to refer to the meta-circular description {section D, page 163} for clarification.

C.1 The Actor Model of Computation

An actor is a fundamental computational entity in the actor model of computation. Computations proceed as actors send communications to other actors, who process them. Each actor has a script, which indicates what communications the actor will accept and how the actor will process each of them. It also has a set of acquaintances, which are the other actors it can communicate with. Notice that each actor contains both data and procedural information (its acquaintances and script). For example, a bank-account actor might have an acquaintance which represents the current balance, and a script which determines how it responds to communications such as deposit or withdrawal requests. This information is encapsulated by the actor, and is therefore hidden from all other actors. The only other actors that an actor can communicate with are its acquaintances and the acquaintances of the incoming communication. Because of this, an actor is not tied down to any hardware processor — it can migrate from machine to machine.

When they receive communications, actors can do simple recognition of the incoming communication, make simple decisions, create new actors, transmit

communications to actors, and change their own behavior. Any actor-based computation is composed from these primitive operations. It is possible to construct sophisticated applications from suitable compositions of actors. Because the actions of actors themselves are inherently concurrent, these applications will also be highly concurrent, with no special effort.

There are very important differences between actors whose scripts do not allow for a change in behavior and actors which may change. An actor which can change its behavior can only process one communication at a time, because the processing of one communication might affect the processing of communications accepted later. Because it must accept communications serially, such an actor is called a *serialized actor*, or *serializer*. It restricts parallelism by requiring synchronization.

Actors which cannot change their behavior are called *unserialized actors*. They can process arbitrarily many communications at a time, requiring no synchronization whatsoever. Moreover, they can be copied indiscriminately when convenient. These actors provide the full potential for parallelism inherent in the actor model.

Communications and the messages within them are also actors. There are three kinds of communications, corresponding to a model of interaction among actors which is analogous to interaction among humans working together on some problem. An actor may send a *request* to another actor, which is often expected to *reply* upon completion of the requested activity or to *complain* if some problem arises. Rather than waiting for a response to a request, which would limit parallelism, an actor spawns new actors to accept the response and pick up with the computation where it left off. These actors are included in addition to the message in the request, so the actor can begin processing the next communication

immediately after sending the request. There are two such actors in each request, a *customer* for processing replies, and a *complaint-department* for processing complaints.

Computation is event-driven. An actor is dormant when not processing a communication. Upon receipt of a communication, it is awakened and can proceed with the computation. This minimizes the resource usage by actors not doing useful work, and makes resource management easier in general.

C.2 A Glimpse of Act2

Actor languages provide a higher-level interface to the basic computational abilities of actors. They can make use of the inherent concurrency of actors to provide concurrency in a natural way at the language level. Act2 is an actor language. A user interacts with it using a listen-loop, one iteration of which reads, parses, and evaluates an Act2 expression, then prints its result.

Act2 has some pre-defined actors, like numbers and symbols. Some standard names are provided, such as `true` and `false`. The symbols are bound to appropriate actors in the standard Act2 environment. Constructors of information structure, such as sequences and instance descriptions, are also pre-defined.

Act2 provides a convenient notation for expressing actor-based computation. For example, the sending of a request and subsequent reception of a response is naturally denoted as an expression in the language. The customer and complaint-department are created by Act2 from information available from the context of the expression.

A convenient form of pattern-matching is provided. It is useful for

recognizing communications and their contents and binding names to their parts for later reference. This pattern-matching unifies the ideas of comparing descriptions and of type-checking.

The evaluation of an expression should always be thought of as sending a request to the expression asking it to evaluate itself in the environment supplied in the request's message. For example, if the expression `(+ 3 x)` is asked to evaluate itself in an environment in which the symbol `x` is bound to `4`, the expression will first ask its sub-expressions to evaluate themselves in the supplied environment. They reply with the values `3` and `4`, then the expression adds them together and replies with the value `7`. In general, if an Act2 construct containing an expression is evaluated and the evaluation of the expression complains, the evaluation of the construct will respond with that complaint, unless the construct explicitly handles that complaint. If there are more than one such complaining sub-expressions, the first to be noticed will be relayed, and the rest will be ignored.

C.3 Pre-Defined Actors

Act2 provides pre-defined actors which can be used in computations. These pre-defined actors correspond to those provided in most other languages: logical truth values, numeric values, symbols, and sequences. All of these are actors, behaving like actors in the computational model. They accept communications, perform some computation as a result, then transmit communications in response. All pre-defined actors are unserialized.

C.3.1 Symbols

A symbol is a name typically used as a keyword or identifier in the Act2 language. In the printed representation of the language, a symbol is denoted by one or more adjacent alpha-numeric characters. For example, **`reply-to`**, **`t1`**, and **`message`** are symbols. Symbols are bound to actors in environments. When asked to evaluate itself as an expression in some environment, a symbol will look itself up in the environment. Symbols also respond to a number of other requests, such as requests to parse, print, or match. Because the evaluation of a symbol is an attempt to get at an actor to which the symbol is bound, denoting the symbol itself is done by using a **quote** expression: **`(quote foo)`** or **`'foo`**.

C.3.2 Numbers

Numbers are actors which behave like numeric mathematical entities. They accept communications such as: a request to add (subtract, multiply, divide, ...) themselves with some other number, or a request to compare themselves with some other number (for equality, or numeric ordering).

C.3.3 Boolean Values

Act2 provides values which behave like logical truth or falsity values. These accept messages such as a request asking them to perform one computation if they represent truth or another if they represent falsity. The identifiers **`true`** and **`false`** are bound to actors with the appropriate behavior.

C.3.4 Sequences

Sequences represent an unserialized, ordered collection of actors. They roughly correspond in behavior to Lisp lists. The concrete realization of that behavior, however, may have many forms. A sequence is created by an expression such as `(sequence` *expressions*`)`, where each expression is a sequence element. Act2 provides a syntactic sugaring for this type of expression: `[`*expressions*`]`. For example, an empty sequence can be denoted `[]`, and a sequence with the elements `3`, `true`, and `-3.14` can be denoted `[3 true -3.14]`. An empty sequence can also be created by the expression `(new empty-sequence)`. `new` expressions will be discussed below.

A non-empty sequence can also be thought of as a recursive data structure, composed of a first element and a sequence containing the rest of the elements. Sequences can be created with an expression of the form,
`(new sequence (with first 3) (with rest ...))`.

C.3.5 Convenient Expression of Basic Operations

Act2 provides convenient expressions for increasing the readability of certain requests such as those handled by pre-defined actors. For example, the expression `(+ 3 4)` is a convenient expression of `(ask 3 (a + (with operand 4)))`.

Other such conveniences include: `(- 3 2)`, numeric subtraction; `(* 3 2)`, numeric multiplication; `(÷ 3 2)`, numeric division; `(∧ true false)`, logical conjunction; `(∨ true false)`, logical disjunction; `(¬ true)`, logical negation; `(= x y)`, equality of arbitrary actors; `(< x y)`, less-than partial ordering; `(> x y)`, greater-than partial ordering, etc.

C.4 Descriptions

When actors perform computations, they cooperate with each other by transmitting communications among themselves. These communications contain messages, which can be arbitrary actors. When an actor receives a communication, it must be able to recognize the communication and the message therein, so it can react to it appropriately, using any information it contains.

Atomic and instance descriptions were developed in [Hewitt, Attardi, Simi 80] for description and reasoning. These descriptions provided a convenient form of expressing and recognizing arbitrary information. Act2 makes use of descriptions for several important and fundamental purposes. Because descriptions can contain arbitrary information in a very convenient way, they are often used as messages in communications, such as `(a deposit (with amount 30))`. They are also used to describe actors, in a way which corresponds roughly to data types in existing languages, such as `(a bank-account (with balance 500))`. Instance descriptions are often used as patterns for recognizing communications, messages, and actors in general, such as `(a deposit (with amount ≡a))`. Patterns and pattern matching will be described in more detail below.

C.4.1 Atomic Descriptions

Atomic descriptions are significant in our descriptions as representations of abstract concepts, such as the concept of **`bank-account`** and of **`deposit`**. They have other uses in Act2, some of which will be described below. Atomic descriptions are often referenced in Act2 expressions representing instance descriptions by a symbol bound to an atomic description. Often, the atomic description has the same name as the symbol used to denote it.

The most convenient way to create new atomic descriptions is with an

expression of the form `(new concept (with name 'foo))`. This creates an atomic description which is distinct from all other atomic descriptions, even from others having the same name. One of the acquaintances of an atomic description (which is hidden by the `concept` interface) is a discriminator which distinguishes it from all others. This discriminator is used in the comparison and matching of atomic descriptions. Two atomic descriptions match if they have the same discriminator.

C.4.2 Instance Descriptions

Instance descriptions abstractly represent a set of instances of some concept. For example, `(a bank-account)` represents the notion of instances of the concept of `bank-account`, and in so doing represents any bank-account which may exist.

Instance descriptions can be specialized by adding further restrictions to what instances they can represent. These restrictions are in the form of *attributes.* For example, `(a bank-account (with balance 500))` represents any bank account having a balance of 500, is significantly more specialized than the description `(a bank-account)`.

Instance descriptions can be used in Act2 for their descriptive capabilities. For example, supposing we had a serialized actor representing a bank account which happens to contain a balance of 500. We could describe this actor as `(a bank-account)` or as `(a bank-account (with balance 500))` as long as those descriptions remain true. Unserialized actors, because they cannot alter their behavior, are even more amenable to description.

Instance descriptions can be used in Act2 for their information-containing capacity. For example, supposing we had a serialized actor representing a bank account with balance 500 dollars and wanted to deposit an additional 30. The most

convenient way to express our desire is to send the bank account a message such as `(a deposit (with amount 30))`.

Instance descriptions can be used in Act2 for their recognition or pattern-matching capabilities. For example, our bank account might be capable of receiving requests to deposit some amount, to withdraw some amount, or to reveal the current balance. The account needs some way to recognize an incoming request, and what it is asking for. It must also be able to extract any additional information from the message, such as the amount to deposit or withdraw. It might use an instance description as a pattern, making use of a few special features for information extraction.

The pattern could look like `(a deposit (with amount ≡a))`. Act2 defines matching such that this pattern would successfully match all specializations of the pattern. This will be described in more detail below. For now, we will simply look at what patterns might look like. The expression, `≡a`, is a convenient way of writing the expression `(bind a)`. If asked to match some actor in an environment E, this expression will bind the identifier `a` to the actor and reply, indicating a successful match, as well as the extended environment. The pattern could impose an additional restriction that the actor be a number:
`(a deposit (with amount (≡a which-is (a number))))`. The pattern could also impose the restriction that the amount be a positive number, as in

```
(a deposit (with amount (≡a which-is (a positive-number)))) or
(a deposit
   (with amount
         (≡a which-is
             ((a number) such-that (> a 0)))))
```

In general, Act2 expressions representing instance descriptions look like:

```
(a concept
   (attribute-kind1 attribute-relation1 attribute-filler1)
   (attribute-kind2 attribute-relation2 attribute-filler2)
   ...)
```

The keyword **an** may be used instead of the keyword **a** as an aid to pronunciation and readability. When asked to evaluate itself in some environment E, an instance description expression will evaluate its *concept* expression and its *attribute-fillers*. It will then create an instance description from the resulting information in addition to its attribute kind and relation information.

Each attribute has a keyword which indicates what kind of an attribute it is. This affects the applicability of various axioms for deduction involving instance descriptions with attributes in Prelude. For the purposes of Act2, which does no sophisticated deduction, the keyword **with** is sufficient for all uses.

Each attribute has a relation name, which indicates the significance of the attribute's filler. By default, Act2 does not evaluate attribute relations, and a raw symbol is sufficient there. This can be thought of as analogous to field names of records in many languages.

Each attribute also has a filler, which contains information of interest. The filler may be a description, or may be an arbitrary actor.

C.4.3 Pattern Matching

Pattern-matching is the fundamental recognition mechanism in Act2. It is used for recognizing a communication and its message, and for binding symbols to some of the parts for later use. This recognition is performed by communication among the actors involved. Typically, an object is available for matching, such as a communication, its message, or some acquaintance. There is also at least one

pattern presented for recognition of the object. Associated with a pattern is some form of processing involving the matched object. For each attempt at a match, we have a pattern for matching, an object to match, and a small environment for holding symbol/actor bindings made during the match. Pattern matching is often a recursive process, first matching the pattern's top level, then matching each of the pattern's fillers.

A typical object to be matched is an instance description or an arbitrary actor. For example, the instance description `(a bank-account (with balance 5))` could be included as a message in a communication, as could any actor, such as a serialized bank-account actor.

A typical pattern is an instance description, which may match another instance description if they are similar, or which may match an arbitrary actor if it is a suitable description of that actor. For example, the following could be used as patterns:

```
(a bank-account)
(a bank-account (with balance ≡b))
(≡x which-is (a bank-account))
```

The pattern-matching performed by Act2 itself does not involve sophisticated deduction based upon knowledge of inheritance relationships among instance descriptions, although constructs providing such matching could be embedded in the language.

Pattern-matching is a negotiation process between patterns and objects, and does not violate the principle of absolute information containment by actors. A pattern-match between a pattern P and an object O is initiated by an Act2 construct by sending P a request with message:

```
(a match
   (with object O)
   (with bindings (new empty-layer)))
```

After some negotiation among the pattern, the object, and their acquaintances, we expect a reply of the form `(a successful-match (with bindings ...))` or `(a failed-match)`.

The behavior of a pattern-match depends upon the way actors used as patterns respond to `match` requests. A few expressions are provided by Act2 which evaluate into actors providing useful functionality for matching. These are often used in conjunction with instance descriptions to construct patterns.

The **bind** expression has the form `(bind` *symbol*`)`, and can be written ≡*symbol*. It does not evaluate its argument. When asked to match some object O, where the set of bindings B has been established, the most common result is for it to simply reply with a successful match, with an extension of B in which the *symbol* is bound to O. It actually checks first whether or not the *symbol* is already bound in B. If not, it simply proceeds as above. If so, the match succeeds (with bindings B) only if the actor bound to the symbol matches the actor, O.

The **which-is** expression has the form `(which-is` *pattern1 pattern2*`)` or (*pattern1* `which-is` *pattern2*). In order for it to result in a successful match, both *pattern1* and *pattern2* must match successfully. A typical use of this expression is to add some restriction to what can be matched by a **bind** expression. For example: `(≡x which-is (a natural-number))`.

A similar expression adds a restriction in the form of a predicate which must be satisfied in order for the match to succeed. A **such-that** expression has the form `(such-that` *pattern predicate*`)` or (*pattern* `such-that` *predicate*). When asked to match some object with established bindings B, it succeeds only if *pattern* succeeds

with some bindings B', and *predicate* yields truth when evaluated in the prevailing environment extended with B'. This might be used in a situation such as: `(≡x such-that (< x 5))`.

Atomic descriptions have a name which is meaningful to humans, and a discriminator, which is actually used to identify them. An atomic description will only match another atomic description which has the same discriminator. Matching atomic descriptions always have the same name. Independently-created atomic descriptions will not match, even if they have the same name, because their discriminators will differ.

An instance description performs a slightly more sophisticated match. For example, the pattern `(a bank-account)` will match both a comparable instance description such as `(a bank-account (with balance 500))` as well as a serialized actor which is described by an instance description such as `(a bank-account)`. Whereas this would naturally occur if Act2 did matching involving deduction, it must be done explicitly by Act2 with an appropriate protocol.

An instance description's simple protocol for matching another instance description is: the concepts must match; the relations present in the pattern must be present in the object; and the fillers in attributes with the same relations must match.

When the object is not a description, the pattern will match the description of the object, rather than the raw object itself. Every actor has a description, which is associated with it at its creation time. Act2's simple protocol for matching two instance descriptions is Act2 has a predicate which can be used to distinguish whether an object matched is an instance description or not. It has the form `(individual` *actor*`)`, and returns truth when applied to actors which are not

descriptions.

Any other actors are provided with extra communication handlers by Act2 if needed, to handle communications such as a request to match. By default, arbitrary actors match if they are "the same actor". Sameness for serialized actors means that the actors must really be the same actor, and must occupy the same storage. Because unserialized actors can be replicated arbitrarily, they are the same if they have the same behavior and the same acquaintances. That is, we cannot tell the difference between copies of an unserialized actor, because their behaviors will never become different.

C.5 Top-Level Expressions

A user's interface to Act2 is a listen loop. At all times, there is a prevailing environment associated with the listen loop. It is with respect to this environment that expressions entered by the user are evaluated.

The user's input is read in as list structure, symbols, and/or numbers. What is read in is asked to parse itself. The resulting abstract syntax is asked to evaluate itself, with respect to the prevailing environment. The response is asked to print itself for the user, then the next iteration begins, prompting the user for more input. Should any unhandled complaints be generated at any point in a listen-loop iteration, the loop itself will handle the exception (by entering a debugger or by asking it to print itself), then will proceed with the next iteration.

The user is able to evaluate arbitrary Act2 expressions simply by typing them in to the listen loop. Act2 provides convenient expressions for: extending the prevailing environment by associating a symbol with an actor denoted by some Act2 expression (**defname**); introducing an abstraction which encapsulates arbitrarily

complex information (**define**); and defining syntactic extensions to the language by extending the environments used in parsing Act2 code (**defexpression** and **defcommand**).

C.5.1 DEFNAME Expression

It is convenient for someone conversing with a listen-loop to remember the results of expression evaluations for later reference. A convenient expression is provided which binds a symbol to an actor in the prevailing environment. For example, `(defname foo (+ 3 4))`, when asked to evaluate itself in some environment, E, would ask `(+ 3 4)` to evaluate itself as an expression in environment E, would accept the reply (7) then ask E
`(a grow (with symbol 'foo) (with value 7))`.

The expression, `(defname` *exp1 exp2*), when asked
`(an expression-eval (with environment E))`, will behave as follows. If all goes well, the environment E grows to associate the symbol *exp1* with the value (V) of the expression *exp2* in E, and the **defname** replies V. If *exp1* is not a symbol, the defname will complain. If *exp2* complains, the defname will relay the complaint.

C.5.2 DEFCONCEPT Expression

Atomic descriptions are a very important part of Act2. Among other things, they serve as concepts for instance descriptions. For flexibility, the concept part of an expression such as `(a foo)` denoting an instance description is evaluated. For readability, it is convenient to express simple concepts simply.

Both constraints are satisfied if the symbol `foo` is bound to a suitable atomic description. The **defconcept** expression is a convenient way of creating an atomic

description and establishing such a binding at the same time. For example, `(defconcept foo)` can be thought of as `(defname foo (new concept (with name 'foo)))`, where `(new concept (with name 'foo))` creates a new atomic description which among other things has the name `foo`. For more details, see {section D.5, page 170} in the meta-circular description of Act2.

C.5.3 DEFINE and NEW Expressions

Act2 has a single abstraction mechanism which is suitable for encapsulating the information content of arbitrarily complex expressions. Only one such mechanism is necessary, because the actor model of computation can express procedural, data, and control abstraction directly in terms of actors. The **define** expression has the form `(define` *expression-template expression*`)`.

Intimately related to **define** expressions are **new** expressions. A **define** declares the meaning of a set of related **new** expressions. For example, `(define (new double (with number ≡n)) (* 2 n))` declares the meanings of a class of expressions including `(new double (with number 3))`, which means `(* 2 3)`, and `(new double (with number -3.14))`, which means `(* 2 -3.14)`. Any expressions with concept `double` but not of the form `(new ... (with number ...))` are undefined, and will complain when evaluated. For more details, see {section D.4, page 168}.

new expressions look very much like instance descriptions, having a concept and optional attributes, but are imperative rather than descriptive. For example, `(new bank-account (with balance 300))` may yield a newly-created bank account with the stated balance, whereas `(a bank-account (with balance 300))` would simply describe such an account. The template in a **define** is typically a **new**

expression whose attribute fillers contain binders. For more details, see {section D.4, page 169}.

A **define** expression of the form `(define` *exp1 exp2*`)` will behave as follows: The abstract syntax *exp1* is asked to install itself, given the prevailing environment and the abstract syntax *exp2*. The **define** expression will relay any unhandled complaints generated by this process, else will acknowledge completion. *exp1* should be a **new** expression.

A later **define** expression declaring a **new** expression with the same concept will shadow the older declaration; only the newest will be used. All concepts in well-formed and meaningful **new** expressions evaluate to atomic descriptions.

C.6 Simple Expressions

C.6.1 ASK Expression

In the actor model, two-way communication is achieved by sending a request containing some message as well as a customer to which the target of the request should reply. The **ask** expression is a convenient way of expressing just that {section D.2, page 165}. In an **ask** expression, a target for the request and the message in the request are explicitly denoted, but the customer is constructed for the user from the context in which the **ask** expression occurs. For example,
`(ask my-bank-account (a balance))` is an expression whose value will be `300` if the symbol `my-bank-account` is bound to an actor which responds to a request with message `(a balance)` with a reply with message `300`.

A useful way of thinking about the **ask** expression is: when asked to evaluate itself as an expression in environment E in a request with customer C, it asks its

target and its message to evaluate themselves in E. It then sends to the target value a request with the message value and the customer C. Therefore, the response from the evaluation of an ask expression is the response from the ask's target when sent the ask's message in a request.

C.6.2 QUOTE Expression

Unless explicitly stated otherwise, Act2 expressions typed in by the user are both parsed and evaluated. This is the case for most contexts in which expressions are expected. Sometimes, it is desirable to be able to denote some unparsed symbol or list structure in a context in which expressions are normally evaluated. The **quote** expression accepts one argument, which it neither parses nor evaluates {section D.2, page 165}. The result of a quote expression is typically either a symbol or some list structure. For example, an evaluation of the expression `(quote foo)` yields the symbol `foo`. An evaluation of the expression `(quote (a 5 (x)))` yields a list containing three elements: the symbol `a`, the number `5`, and a list with one element, the symbol `x`.

A prefix operator (`'`), known as "quote", "single-quote", or "accent-mark", is provided for convenience. The expression `'`*exp* is syntactically equivalent to the expression `(quote` *exp*`)`. Thus, the examples above could have been written `'foo` and `'(a 5 (x))`.

C.6.3 PARSE-EXPRESSION and PARSE-COMMAND Expressions

The **parse-expression** expression parses its argument as an expression, producing an abstract syntax actor. It is included only for convenience, because its effect can be reproduced by sending a parse request to a quoted expression. An expression such as `(parse-expression '(ask foo (a decrement)))` evaluates to

an abstract syntax actor representing the expression `(ask foo (a decrement))`, which might later be asked to evaluate itself in some environment. A similar expression, **parse-command**, exists for parsing surface syntax into abstract syntax representing a command.

C.7 Creating Actors

The **create** expression provides a mechanism for creating actors having a specific behavior. It contains communication handlers describing what messages the actor will accept and how the actor will react to each of them. Each communication handler has two parts. A matching part contains an instance-description pattern, which characterizes the communications the handler will accept and extracts information from communications it matches. A body part contains a set of commands to be evaluated when a communication is accepted by the handler. Act2 commands will be described below.

When a **create** expression is asked to evaluate itself in some environment, it constructs a new actor from that environment and the communication handlers {section D.7, page 172}. It is useful to think of this actor as if it retains the creation environment and the abstract syntax for the handlers. At times, the evaluation message may contain extra information, such as a description of the actor. This information is incorporated in the newly-created actor, as will be mentioned in the discussion of the **define** and **new** expressions. The newly-created actor is capable of accepting and processing communications, as dictated by the communication handlers.

The most common communication handler is for accepting requests, and has the form `(is-request` *message-pattern* `do` *commands*`)`. If the incoming

communication is a request, and its message is an actor which matches the *message-pattern*, then the communication handler is capable of accepting the communication. There are similar forms of communication handlers for accepting replies, **(is-reply** *message-pattern* **do** *commands*), and complaints, **(is-complaint** *message-pattern* **do** *commands*). There is also a more general form of communication handler, which allows explicit extraction of more or less information from the communication. It contains a pattern for matching the whole communication, rather than just its message:

(is-communication *communication-pattern* **do** *commands*).

A create expression itself has the following form, where an **otherwise** clause can be omitted at any point:

```
(create
 communication-handlers
 (otherwise communication-handlers
            (otherwise communication-handlers
                       (otherwise ...))))
```

Upon receipt of a communication, all handlers in the first set of handlers are given a chance to match the incoming communication. These attempts at matching are performed concurrently. If any of the handlers successfully matches the communication, one is chosen (the first one noticed, temporally) to handle the message. If all attempts at matching by these handlers fail, some handlers supplied by Act2 will be tried by default. These handle such communications as requests to print or requests to match some actor. If these fail, and there is an **otherwise** clause, the next set of handlers is tried. This process continues until a handler is found that can accept the communication, or until there are no more handlers to try. In the latter case, the actor rejects the communication.

Rejecting a communication happens as follows. If the communication is a request, the actor complains to the complaint-department designated in the request;

otherwise, the actor complains to a standard complaint-department reserved for such purposes by the implementor.

A communication handler chosen to process the communication evaluates the commands in its body, using the actor's creation environment, extended with any bindings established during the match {section D.10.6, page 195}. The commands in its body are evaluated concurrently. Some important pieces of information, such as a customer, complaint department, or sponsor, are often left un-named or completely unmentioned in communication handlers. Act2 has context-sensitive commands which can make use of this information.

In principle, the create operation is sufficient for creating actors. We have two kinds of actors: serialized and unserialized. Serialized actors are able to change their behavior, and are therefore not permitted to handle more than one communication concurrently. Unserialized actors can not only handle many communications concurrently, but can also be replicated indiscriminately. The distinction between them is important not only for performance, but for recursive computations[4]. By default, **create** creates a serialized actor, to be on the safe side, because an interpreter does not conveniently know whether or not one of the handlers with cause a change in behavior. A compiler, on the other hand, could create unserialized actors from a create expression when it notices that the actor's behavior cannot change.

---

[4]Because a serialized actor can only process one communication at a time, it cannot send itself communications as part of the processing of another communication. This would freeze the actor in a deadlock. For example, consider our bank account example from {section 2.3, page 34}. Suppose we had implemented the serialized actor to respond to a deposit request by sending itself a withdrawal request with the negated amount. Because the serialized actor can only process one communication at a time, The withdrawal request would have simply been enqueued for the actor to process later. Because it will never get a response from its withdrawal request, the deposit request will never be satisfied.

Act2 provides the **create-unserialized** expression, which behaves like the create expression, but always creates unserialized actors. This is an aid to the interpreter, which does not have enough information conveniently at its disposal to deduce that the actor cannot change. It is an optimization for compilation, saving the work it would otherwise take to determine that the actor is unserialized. It is also good documentation for human readers of Act2 code. No attempts in a communication handler body to change the behavior of one of these actors will be honored, and a complaint will be generated as soon as this is noticed.

A **create-unserialized** expression itself has the following form, where an **otherwise** clause can be omitted at any point:

```
(create-unserialized
 communication-handlers
 (otherwise communication-handlers
            (otherwise communication-handlers
                       (otherwise ...))))
```

By default, the descriptor which is associated with a newly-created actor contains no information about the actor's state. For example, if a bank account abstraction was defined with

```
(define (new account (with balance ≡b))
  (create ...))
```

accounts created with expressions of the form `(new account (with balance 500))` would be associated with the description `(an account)`, instead of `(an account (with balance 500))`. This helps guarantee the opacity of these actors, since the balance could not be obtained with simple pattern-matching. The implementor of an abstraction may make this information available by default by using other variations of the **create** expression. The **create-visible** and **create-visible-unserialized** expressions do exactly this, and have syntax similar to that of the **create** and **create-unserialized** expressions. In order to make the balance of these bank accounts available for pattern-matching, the accounts would have been

defined with **define** and **create-visible** expressions of the form

```
(define (new account (with balance ≡b))
  (create-visible ...))
```

C.8 Simple Context-Free Commands

Few of Act2's commands are completely context-free. They are for one-way transmission of communications, where both the communication and target are fully specified. These commands can be included in any context where commands can be put.

C.8.1 REPLY-TO Command

The reply-to command specifies a target and a message. When successfully evaluated, it creates a reply communication containing the message, then transmits that communication to the target {section D.8, page 174}. For example, `(reply-to customer 3)` sends a reply communication with message 3 to the actor bound to the symbol `customer` in the evaluation environment.

More specifically, the form of the **reply-to** command is `(reply-to` *target message*`)`. When asked to evaluate itself as a command in some environment E, it asks *target* and *message* to evaluate themselves as expressions in environment E. If they both reply, it transmits a reply communication containing the message value to the target value. If *target* complains, then the **reply-to** command relays the complaint. Otherwise, if *message* complains, the complaint is transmitted to the target value.

C.8.2 COMPLAIN-TO Command

The complain-to command specifies a target and a message. When successfully evaluated, it creates a complaint communication containing the message, then transmits that communication to the target. For example, `(complain-to complaint-department (a failure))` sends a complaint communication with message `(a failure)` to the actor bound to the symbol `complaint-department` in the evaluation environment.

More specifically, the form of the **complain-to** command is `(complain-to` *target message*`)`. When asked to evaluate itself as a command in some environment E, it asks *target* and *message* to evaluate themselves as expressions in environment E {section D.8, page 174}. If they both reply, it transmits a complaint communication containing the message value to the target value. If *target* complains, then the **complain-to** command relays the complaint. Otherwise, if *message* complains, that complaint is transmitted to the target value.

C.8.3 SEND-TO Command

The **send-to** command is for transmitting an arbitrary communication to some specified target. It is similar in behavior to the **reply-to** and **complain-to** commands, except that the whole communication to be transmitted is specified, rather than just the message {section D.8, page 174}. The examples above could have been written as `(send-to target (new reply (with message 3)))` and `(send-to target (new complaint (with message (a failure))))`.

C.9 Composite Constructs

Concepts such as name-binding, decision, and complaint-handling are useful both in the context of commands and expressions. Act2 provides such a set of constructs which can be used both as commands and as expressions.

The evaluation of each of these constructs may involve the evaluation of a body of commands included in the construct. The commands allowed in the bodies, and the meaning of a few of those commands, depend upon the context in which the construct appears. The construct may be used as an expression, as a command in a communication-handler body, or as a command in an expression body. This will be explained in more detail below.

C.9.1 LET Construct

The let construct allows the extension of the evaluation environment with symbol-actor bindings resulting from one or more attempts at matching. The extended environment is used in the evaluation of the commands in the body of the **let** construct {section D.9.3, page 181}.

An example of the use of a **let** command is:

```
(let ((≡x match 3)
      (≡y match (+ 2 2))
      ((a deposit (with amount ≡z))
       match incoming-message))
  do
  (reply (+ x y))
  (become (new frotz (with balance z))))
```

In general, the form of let constructs is:

```
(let ((pattern1 match expression1)
      (pattern2 match expression2)
      ...)
  do
  command1
  command2
  ...)
```

When evaluated in some environment E, as either a command or as an expression, the set of matchers is first processed. The patterns and expressions in the matchers are evaluated concurrently in the environment E. If any complain, the let construct relays that complaint. Next, each pattern is asked to match the corresponding expression. If any of the matches are not successful, the let construct complains. Otherwise, E is extended with all bindings made during the matchings, then the commands in the body are evaluated concurrently in the extended environment. If the evaluation of any of these commands complains, then the let construct relays the complaint.

C.9.2 LABEL Expression

The **label** expression is introduced for convenience in denoting self-reference. For example, wrapped around a **create** expression, it allows an actor to reference itself with a locally-bound identifier. The **label** expression has the form

(label *symbol expression*)

It is essentially equivalent to the expression

```
(let ((≡symbol match (delay expression))) do
  (reply symbol))
```

C.9.3 Interpretation of Command Bodies

As mentioned earlier, there are restrictions on the commands allowed or required in command bodies, depending upon usage of the construct. If the construct is used as a command, the commands allowed in the construct's body are

the same as the commands allowed in the context in which the construct exists. Those commands mean the same as they would if they had occurred in the context in which the construct exists. This will become clear as we describe composite constructs in more detail below. For example, in the command body of a communication handler, we can have a **become** command, **reply** or **complain** commands, and others. If one of our commands is a **let** construct, its command body can contain exactly those commands which were allowed in the context in which the **let** construct appeared. That is, it can contain a **become** command, **reply** or **complain** commands, and more. The meaning of and restrictions on the commands in its command body are the same as if those commands had appeared instead of the **let** construct.

If the construct is used as an expression, there are different restrictions on the commands which can appear in its command body, and **reply** and **complain** commands have a different and special meaning. The evaluation of the construct must include the evaluation of a single command which denotes the value of the expression with a **reply** command, or which generates a complaint with a **complain** command. These commands will be described below.

C.9.4 ONE-OF Construct

The basic decision-making construct in Act2 is the **one-of** construct. This has the form, where the **otherwise** clause may be omitted at any stage:

```
(one-of
 (if expression do commands)
 ...
 (otherwise (if expression do commands)
            ...
            (otherwise ...)))
```

When asked to evaluate itself, the construct concurrently evaluates the

expressions in its first set of arms {section D.9.2, page 179}. Of those returning an actor behaving like the truth value **true**, one is chosen, and the body of commands it guards is evaluated. If any of the expression evaluations complains, the construct complains. If any of the expressions yields an actor which does not behave like a truth value, the construct complains. If all expressions yield an actor behaving like the truth value **false**: if there is an **otherwise** clause with another set of guards, then the above process is repeated; if there is no **otherwise** clause, the construct complains. If a body of commands is chosen for evaluation and its evaluation causes at least one unhandled complaint, then the **one-of** construct will relay the first complaint it notices.

C.9.5 IF Construct

Act2 provides a convenient construct for simple two-way decisions, the **if** construct. It has the form

```
(if expression
  (then do commands)
  (else do commands))
```

As expected, this is simply a convenient form of writing

```
(one-of
 (if expression do commands)
 (if (¬ expression) do commands))
```

C.9.6 CASE-FOR Construct

Another composite construct is used for handling the result of evaluating an (arbitrarily complex) expression. It allows the pattern-matching of the message from the evaluation's reply or complaint, followed by the evaluation of some body of commands associated with the winning matcher. The match can involve binding symbols to parts of the incoming message, which will be used in the evaluation of

the chosen body. The pattern-matching itself provides a form of decision-making.

The case-for construct has the form:

```
(case-for expression
  response-handlers
  (otherwise response-handlers
             (otherwise ...)))
```

When asked to evaluate itself, it evaluates the expression, whose result will be matched {section D.9.1, page 176}. This will result in either a reply or a complaint communication with some message. Correspondingly, there are two types of response-handlers, one for matching reply communications, `(is` *message-pattern* `do` *commands*`)`, and one for complaint communications, `(complaint` *message-pattern* `do` *commands*`)`. When a handler is involved in the matching process, the following happens: if the type of communication is incompatible, the match fails; otherwise, the expression denoting a pattern is evaluated, yielding a pattern. If the evaluation complains, the whole construct relays the complaint. Next, the pattern is asked to match the message from the incoming communication.

The first set of response-handlers is checked concurrently for those capable of handling the reply or complaint communication. The first one noticed which can handle the communication is chosen, and the commands in its body are evaluated in the evaluation environment for the construct extended with any bindings established in the pattern-match. If all of these attempts at matching fail: if there is an **otherwise** clause, the matching attempt is continued; if there is none, and the communication being matched is a complaint, that complaint is relayed, otherwise a standard complaint is generated.

C.10 Context-Sensitive Commands

Act2 provides a few commands whose meaning depends upon the context in which they appear. There are two major contexts in which commands appear: in the bodies of composite expressions such as **let**, **one-of**, or **case-for** expressions (the *expression-body-context*); and in the bodies of communication handlers in expressions such as **create** or **create-unserialized**, which describe the behavior of actors (the *handler-body-context*).

C.10.1 REPLY Command

The reply command represents the transmission of a specified reply communication to some unspecified target. The target of the reply depends upon the context in which the reply command occurs. The reply command has the form `(reply` *expression*`)`. When asked to evaluate itself, it asks the expression to evaluate itself, then sends the result as a message in a reply communication to the unspecified target {section D.8, page 174}. Examples of this will appear below. Should the evaluation of the expression complain, that complaint is transmitted to prevailing complaint-department instead. The target of the reply depends on the context in which the **reply** command occurs.

If the **reply** command occurs in a handler-body-context, the behavior depends upon the type of communication received. The **reply** command is intended to be used only when handling request communications, which contain a customer and complaint department. If this is the case, the reply will be sent to the customer. Should any problems occur in evaluation, the resulting complaint will be relayed to the complaint department. In the event that the incoming request is not a request, some implementation error exists, so a complaint is sent to the implementor of the actor. For example, consider a **strange-actor** abstraction defined as

```
(define (new strange-actor)
  (create
    (is-request ≡m do (reply 3))
    (is-complaint ≡m do (reply 4))))
```

If such an actor receives a request, it will transmit a reply with message **3** to the customer included in the request. Notice that this customer is not mentioned anywhere in the Act2 implementation of the actor. If such an actor receives a complaint, the **reply** command, not having a customer to which to reply, will instead complain of an implementation error.

If the **reply** command occurs in an expression-body-context, we think of the evaluation of the command as occurring in response to a request for evaluation of the expression. Therefore, the reply is sent to that request's customer. Should a complaint occur in the evaluation, it is relayed to the request's complaint--department, as usual. The net effect of this is that the **reply** command denotes the value of the expression. For example, consider the expression

```
(+ 5
   (let ((≡x match 3)
         (≡y match 4)) do
     (reply (* x y))))
```

The **let** construct is used as an expression. The **reply** command in its body indicates that the construct will reply with a **12** when asked to evaluate itself as an expression. The **+** expression will therefore reply with a **17** when asked to evaluate itself as an expression.

Exactly one **reply** or **complain** command must be encountered in the evaluation of a composite expression's body. If neither is evaluated, or more than one is evaluated, then a complaint is generated.

C.10.2 COMPLAIN Command

The **complain** command is similar to the **reply** command, and has the form `(complain` *expression*`)`. Rather than sending a reply with the denoted actor as its message, the complain command transmits a complaint containing it instead. In essence, the **complain** command indicates that the evaluation of this expression should result in a complaint.

C.10.3 BECOME Command

The **become** command may occur in the body of a communication handler in an expression which creates an actor. We have already seen an example of this in the `account` example {section 2.3, page 34}, reproduced below with the **become** commands in bold italics.

```
(define (new account
             (with balance ≡b))
  (create
   (is-request (a balance) do (reply (a balance)))
   (is-request (a deposit (with amount ≡a)) do
     (become (new account (with balance (+ b a))))
     (reply (a deposit-receipt (with amount a))))
   (is-request (a withdrawal (with amount ≡a)) do
     (let ((≡new-balance match (- b a))) do
       (if (≥ new-balance 0)
         (then do
           (become (new account (with balance new-balance)))
           (reply (a withdrawal-receipt (with amount a))))
         (else do
           (complain (an overdraft))))))))
```

It has the form `(become` *expression*`)`, where *expression* denotes a replacement actor. The become command may be evaluated when the actor accepts a communication. When it is evaluated, it first asks the expression to evaluate itself. If the expression evaluation complains, that complaint is relayed through the **become** command. Otherwise, the actor changes its behavior such that it is indistinguishable from the replacement actor resulting from the evaluation of the

expression in the become command. For more details, see {section D.8, page 173} and {section D.10.4, page 189}.

No more than one become command should be evaluated in the evaluation of a handler body. If an attempt to do this is made, a complaint will be generated. The become command is not permitted in an expression-body-context. If it does occur there, a complaint will be generated. For example, the **become** command would be inappropriate in a context such as

```
(+ 5
   (let ((≡x match 3)
         (≡y match 4)) do
     (become (* x y))
     (reply (* x y))))
```

If the actor was created with a **new** expression declared by a **define** expression of the form `(define (new ...) (create...))`, the expression in a become command in one of its handlers can specify values for some of the attributes, and fillers for the rest will be derived from the actor itself. That is, only those attributes which are different need be mentioned. For example, the new expression in the become command below is equivalent to

```
(new checking-account (with balance ...) (with owner o))
```

```
(define (new checking-account (with balance ≡b) (with owner ≡o))
  (create
   (is-request (a deposit ...) do
     (become (new checking-account (with balance ...)))
     ...)
   ...))
```

C.11 Other Commands

C.11.1 CONCURRENT and SEQUENTIAL Commands

The **concurrent** and **sequential** commands have the form `(concurrent` *commands*`)` and `(sequential` *commands*`)`. Commands are normally evaluated concurrently in Act2, so unless nested inside a sequential command, a **concurrent** command serves only to group a set of commands into a single command. This is useful in conjunction with the **handle-complaints** command, which is described below. The net effect of a **concurrent** command is to cause the concurrent evaluation of the commands it contains. The **sequential** command, on the other hand, causes the commands to be evaluated sequentially, in order of occurrence. If any of the commands in either should complain, then the **sequential** or **concurrent** command simply relays the first complaint it notices.

C.11.2 HANDLE-COMPLAINTS Command

The case-for construct is useful for handling complaints generated in the evaluation of an expression. It is also useful to be able to handle complaints generated in the evaluation of a command. The **handle-complaints** command does exactly this. For example, if we wish to handle some of the complaints which might arise in the body of a communication handler for some actor, we might define the actor as

```
(define (new foo ...)
  (create
    (is-request ... do
      (handle-complaints command
        (complaint (a bar ...) do ...)
        ...))
    ...))
```

or if we wish to handle complaints from more than one command, we can group those commands with a **concurrent** command, as in

```
(define (new foo ...)
  (create
    (is-request ... do
      (handle-complaints (concurrent commands)
        (complaint (a bar ...) do ...)
        ...))
    ...))
```

It looks very similar to the **case-for** command. Rather than guarding an expression, it guards a command. Rather than having both **is** and **complaint** handlers, it has only **complaint** handlers, since commands do not reply with a value. Therefore, this command has the form, where any of the **otherwise** clauses may be omitted:

```
(handle-complaints command
  (complaint pattern do commands)
  ...
  (otherwise (complaint pattern do commands)
             ...
             (otherwise ...)))
```

If no complaints are generated by *command* the meaning of the **handle-complaints** is the same as the meaning of *command* itself. If a complaint is generated in the evaluation of *command*, then the behavior of the **handle-complaints** is very similar to that of a **case-for** construct used as a command. Each set of complaint handlers will be tried in turn.

C.11.3 USING-SPONSOR Construct

Act2 makes use of special actors known as sponsors for resource management, in order to impose some control over parallel computation. Normally, users need not bother with these, since the default policies for resource usage are adequate for average use. Programmers desiring more control over resource usage by different commands or expressions may make use of the **using-sponsor** construct. It has the form `(using-sponsor` *expression* `do` *commands*`)`.

All communications transmitted have a sponsor as an acquaintance. By default, this sponsor provides the resources for the computation performed upon acceptance of the communication. This sponsor can be bound to an identifier when matching the communication, then used later in a **using-sponsor** construct.

When the construct is evaluated, it evaluates *expression*, which should denote a sponsor. This sponsor is used to provide resources for the evaluation of commands in the body of the construct. The **using-sponsor** construct will relay any complaints generated in the evaluation of the expression or of the commands.

C.11.4 Comments

Comments can be inserted anywhere in Act2 code where separators such as space characters or other white-space can occur. A comment begins with a semi-colon (;) and ends with the next end-of-line character. Any sequence of characters can occur between these.

C.12 Syntactic Extension

In Act2, user input is read in as list structure and symbols. Whatever is read in is asked to parse itself as either an expression or a command, and is provided with two special keyword environments. Symbols (and numbers) ignore the environments and parse themselves directly, but a list or sequence will scan itself from left to right, looking for a symbol which has been declared as a keyword. Mechanisms for declaring such keywords will be presented below.

Each keyword environment associates a symbol with an actor which parses sequences. When a list or sequence is asked to parse itself as an expression, it scans itself from left to right. Whenever it encounters a symbol, it looks that symbol up in

the expression-keyword environment. If the symbol is not a keyword, the scan continues. If it is a keyword, the parser to which it is bound is asked to parse the sequence.

Users share common parsing environments for the basic Act2 language definition. In addition, each user's environment has private environments for personal extensions. These personal environments are bound to the identifiers, `standard-act2-expressions` and `standard-act2-commands`.

The **defexpression** and **defcommand** expressions are included in Act2 for convenient extension of these personal parsing environments. Syntactic extension is meant less for casual users than for language designers embedding new languages in Act2.

C.12.1 DEFEXPRESSION Expression

The **defexpression** expression has the form `(defexpression` *symbol exp*`)`. When asked to evaluate itself, the expression proceeds with the evaluation of *exp*, which should be an expression denoting or creating an actor which will parse a sequence identified with the *symbol*. It then asks the default expression-keyword environment to extend itself with a mapping from the *symbol* to the parser. If any unhandled complaints are generated in these attempts, they are relayed as the response from the evaluation of the **defexpression**.

Here is an example of the use of **defexpression** to declare a new kind of expression with the form `(delay` *expression*`)`, which will provide a lazy evaluation capability. When evaluated, the **delay** expression will return immediately with a newly created `delay` actor, without evaluating *expression*. If and when the `delay` actor is ever sent a communication, the delayed *expression* will be evaluated, and the

result will be sent the communication.

Here we establish the expression keyword, and install an appropriate parser. The full implementation of the parser is shown, to illustrate the various messages which get passed around. In practice, there would be a set of parameterized parser abstractions available, and only a simple instantiation of one of them would be required.

```
(defexpression delay
  (create
   (is-request (an expression-parse
                   (with source ≡src)
                   (with expression-keywords ≡ek)
                   (with command-keywords ≡ck))
               do
     (case-for src
       (is ['delay ≡exp] do
         (reply
          (new delay-expression
               (with arg
                 (ask exp (a parse-yourself-as-expression
                             (with expression-keywords ek)
                             (with command-keywords ck)))))))))))
```

Here is an implementation of an abstract syntax actor abstraction for representing **delay** expressions.

```
(define (new delay-expression (with arg ≡exp))
  (create
   (is-request (an expression-eval (with environment ≡env)) do
     (reply
      (create ;; a serializer representing the evaluated expression.
       (is-communication ≡com do
          ;; if it ever gets a communication,
          ;; we should eval the expression in original environment.
          (case-for (ask exp (an expression-eval
                                (with environment env)))
            (is ≡value do   ;; if the evaluation succeeds:
              (send-to value com) ;; send the communication to it, and
              ;; become the result, so future communications go
              ;; directly to it.
              (become value))
            (complaint ≡reason do  ;; if the evaluation fails:
              (complain reason)          ;; relay the complaint, and
              ;; become something that will complain in the same way
              ;; to any further communications.
              (become (create-unserialized
                        (is-communication ≡x do
                          (complain reason)))))))))))
```

Here is a simple example of the creation of an expression, `(prevailing-environment)`, whose value is the current environment, in which the expression itself is evaluated. A single actor serves both as parser and as abstract syntax.

```
(defexpression prevailing-environment
  (label ≡self
    (create-unserialized
     (is-request (an expression-parse) do (reply self))
     (is-request (an expression-eval
                    (with environment ≡e))
                 do (reply e)))))
```

C.12.2 DEFCOMMAND Expression

The **defcommand** expression is identical to the **defexpression** expression, except that it establishes a new command keyword, rather than a new expression keyword.

Appendix D

A Meta-Circular Description of Act2

This appendix contains a meta-circular description of Act2. It consists of Act2 implementations of abstract syntax objects representing the expressions and commands in Act2. Our preliminary Scripter implementation of Act2 was very closely patterned after this description.

To save typing space, an expression of the form `(evaluate` *exp env*`)` was introduced. It means exactly the same as
`(ask` *exp* `(an expression-eval (with environment` *env*`)))`.

This description relies on some aspects of Act2 which increase the conciseness of the code. For example, the evaluation of an expression will directly relay a complaint in the evaluation of one of its sub-expressions. Unhandled communications will result in a complaint. These and similar cases are explicitly included in the Scripter implementation.

D.1 Primitive Actors

Primitive actors are implemented in cooperation with the underlying apiary. They are represented directly as the corresponding Lisp objects. Primitive scripts are associated with each category of primitive actors. Their meta-circular descriptions are simply a high-level representation of their behavior.

Numbers include integers and reals, both positive and negative. They are represented by Lisp fixnums, flonums, and bignums.

```
(define (new number-expression (with value ≡v))
 (label self
  (create-unserialized
   (is-request (an expression-eval) do (reply v))
   (is-request (a match (with bindings ≡b) (with object ≡o)) do
     (case-for o
       (is self do (reply (a successful-match (with bindings b))))
       (otherwise (is something do (reply (a failed-match))))))
   (is-request (a zerop) do (reply (= v 0)))
   ;; similarly: plusp minusp oddp minus abs 1+ 1- fix float
   (is-request (a + (with operand ≡x)) do (reply (+ x v)))
   ;; similarly: = > >= < <= max min - * // remainder gcd
   ...)))
```

Symbols serve as keywords and identifiers in Act2. In the underlying implementation, `T` also represents the logical value of truth, and `NIL` represents the logical value of falsity. `NIL` also serves as an empty list.

```
(define (new symbol-expression (with symbol ≡s))
 (label self
  (create-unserialized
   (is-request (an expression-eval (with environment ≡env)) do
     (reply (ask env (a lookup (with symbol s)))))
   (is-request (an install (with creation-expression ≡ce)
                           (with environment ≡env))        do
     (case-for (ask ce (an expression-eval (with environment env)))
       (is ≡v do
         (reply (ask env (a grow (with symbol s) (with value v))))))))
   ...)))
```

A sequence represents an ordered collection of actors. It can be used as a pattern, with bind-expressions for elements. By default, sequences are represented as Lisp lists.

```
(define (new sequence (with first ≡f) (with rest ≡r))
 (create-unserialized
  (is-request (≡eval which-is (an expression-eval)) do
      (reply (new sequence (with first (ask f eval))
                           (with rest (ask r eval)))))
  (is-request (a first) do (reply f))
  (is-request (a rest) do (reply r))
  (is-request (a length) do (reply (+ 1 (ask r (a length)))))
  (is-request (a match (with bindings ≡b) (with object ≡o)) do
      (case-for o
        (is (a sequence (with first ≡o1) (with rest ≡or)) do
            (case-for (ask f (a match (with bindings b) (with object o1)))
               (is (a successful-match (with bindings ≡b)) do
                  (reply (ask r (a match (with bindings b)
                                         (with object or)))))
               (otherwise (is something do (reply (a failed-match)))
                          (complaint something do (reply (a failed-match))))))
        (otherwise (is something do (reply (a failed-match))))))
  ...))
```

D.2 Simple Expressions

The **quote** expression simply prevents the parsing and evaluation of an expression.

```
;; expression: (quote expression)
(define (new quote-expression (with source ≡s))
 (create-unserialized
  (is-request (an expression-eval) do (reply s))
  ...))
```

The **ask** expression represents the sending of a request to some target, and the receipt of a response. It looks to the programmer like a two-way communication exchange.

```
; expression (ask target message)
(define (new ask-expression
             (with target ≡t)
             (with message ≡m))
  (create-unserialized
   (is-request (≡eval which-is (an expression-eval)) do
     (reply (ask (ask t eval) (ask m eval))))
   ...))
```

A convenient expressional notation is provided for primitive operations, such as addition, subtraction, and conjunction. A simple protocol is used, so these

operations are all represented with the same ae abstract syntax object.

```
; expression: (+ 1 2) (1 + 2) ...
(define (new binary-operator
             (with operator ≡op)
             (with lhs ≡left)
             (with rhs ≡right))
 (create-unserialized
  (is-request (≡eval which-is (an expression-eval)) do
    (reply (ask (ask left eval)
                (an op (with operand (ask right eval))))))
  ...))
```

The **delay** expression provides the ability to perform lazy evaluation on demand.

```
(define (new delay-expression
             (with expression ≡exp))
  (create-unserialized
    (is-request (≡eval which-is (an expression-eval)) do
      (reply
        (create
          (is-communication ≡com do
            (case-for (ask exp eval)
              (is ≡value do
                  (send-to value com)
                  (become value))
              (complaint ≡msg do
                  (complain msg)
                  (become
                    (create-unserialized
                      (is-communication something do
                        (complain msg)))))))))))
```

D.3 Variable Binding

Act2 has a few special expressions for use as patterns. One is for binding an identifier to the corresponding actor in the object of the match. Others put restrictions on the actors which can be bound.

The **bind** expression is used as a pattern, to bind an identifier to the actor it is supposed to match. If the identifier has not been bound during the match, or if it has been bound to the same actor as the one being matched, the match succeeds. Otherwise, the match must fail.

```
; expression: "≡x", "(bind x)"
(define (new bind-expression (with symbol ≡s))
 (label self
  (create-unserialized
   (is-request (an expression-eval) do (reply self))
   (is-request (a match (with bindings ≡b) (with object ≡o)) do
     (case-for (ask b (a lookup (with symbol s)))
       (complaint ≡m do
         (reply (a successful-match
                   (with bindings
                         (ask B (a grow
                                   (with symbol s)
                                   (with value o)))))))
       (is ≡v do (case-for o
                  (same v do (reply (a successful-match (with bindings b))))
                  (otherwise do (reply (a failed-match))))))
   ...)))
```

The **which-is** expression is usually used to place restrictions on what a **bind** expression can match. Essentially, the **which-is** expression is a binary conjunction operator for descriptions.

```
; expression: "(which-is ≡x PATTERN)"
(define (new which-is-expression (with lhs ≡l) (with rhs ≡r))
  (create-unserialized
   (is-request (≡ which-is (an expression-eval)) do
     (reply (new which-is-expression (with lhs (ask l eval))
                                     (with rhs (ask r eval)))))
   (is-request (a match (with bindings ≡b) (with object ≡o)) do
     (case-for (ask l (a match (with bindings b) (with object o)))
       (is (a successful-match (with bindings ≡b1)) do
         (reply (ask r (a match (with bindings b1) (with object o)))))
       (otherwise (is something do (reply (a failed-match))))))
   ...))
```

The **such-that** expression provides another way to filter a pattern-match. It provides a description to match, as usual. It also provides a predicate which decides whether or not to let the match succeed after the match with the description has succeeded.

```
(define (new such-that-expression
            (with description ≡d)
            (with predicate ≡p))
  (create-unserialized
   (is-request (≡eval which-is (an expression-eval
                                  (with environment ≡env))) do
     (let ((≡desc match (ask d eval))) do
       (create-unserialized
        (is-request (≡msg which-is (a match
                                       (with bindings ≡b)
                                       (with object ≡o)))    do
          (case-for (ask desc msg)
            (is (a failed-match) do (reply (a failed-match)))
            (is (≡result which-is (a successful-match
                                      (with bindings ≡b))) do
              (if (evaluate p (new environment
                                   (with primary b)
                                   (with secondary env)))
                (then do (reply result))
                (else do (reply (a failed-match))))))))))))
```

D.4 Abstraction

Act2 has a single abstraction mechanism. There are two aspects of the abstraction mechanism: definition of an abstraction and instantiation.

The **define** expression is for defining a new abstraction. The expression contains a template (or "pattern") characterizing a set of **new** expressions, and another expression which denotes a meaning for those **new** expressions.

```
; expression: (define creation-template abstracted-expression)
(define (new define-expression
            (with creation-template ≡t)
            (with abstracted-expression ≡ce))
 (create-unserialized
  (is-request (an expression-eval (with environment ≡e)) do
    (reply (ask t (an install
                      (with expression ce)
                      (with environment e))))
  ...))
```

The **new** expression is for instantiating abstractions.

```
(define (new new-expression
             (with concept ≡c)
             (with attribute-sequence ≡as))
 (create-unserialized
  (is-request (an install (with expression ≡ce)
                          (with environment ≡e))          do
    (case-for (ask e (a lookup (with symbol c)))
      (is (≡ad which-is (an atomic-description)) do
        (reply (ask ad (an install-implementation
                         (with environment e)
                         (with expression ce)
                         (with pattern
                           (evaluate (new instance-description
                                          (with concept c)
                                          (with attribute-sequence as))
                                     e))))))))
  (is-request (≡eval which-is (an expression-eval)) do
    (case-for (ask c eval)
      (is (≡ad which-is (an atomic-description)) do
        (case-for (ask ad (a summarize-implementation))
          (is (an installation (with environment ≡e1)
                               (with expression ≡ce1)
                               (with pattern ≡cp1))     do
            (let ((≡state match
                    ;; There's actually a bit more to it than this.
                    (ask (new instance-description
                              (with concept c)
                              (with attribute-sequence as))
                         eval)))
              do
              (case-for (ask cp1 (a match
                                   (with bindings (new empty-layer))
                                   (with object state)))
                (is (a successful-match (with bindings ≡b)) do
                  (reply (ask ce1
                              (an expression-eval
                                  (with environment
                                        (new environment
                                             (with primary b)
                                             (with secondary e1)))
                                  (with pattern cp1)
                                  (with state state)))))))))))
 ...))
```

D.5 Extending Listener's Environment

The **defname** expression is used to associate a name with the result of evaluating an expression, at the top level listener.

```
(define (new defname-expression
             (with symbol ≡sym)
             (with expression ≡exp))
  (create-unserialized
   (is-request (≡eval which-is (an expression-eval
                                    (with environment ≡e))) do
     (reply (ask e (a grow
                      (with symbol sym)
                      (with value (ask exp eval))))))))
```

The **defconcept** expression is for extending the prevailing environment with a new concept. It is meant to be used at top level, from a listen-loop.

```
(define (new defconcept-expression (with symbol ≡s))
 (label self
  (create-unserialized
   (is-request (a expression-eval (with environment ≡e)) do
     (let ((something match
                (ask e (a grow
                          (with symbol s)
                          (with value (new concept (with name s)))))))) do
          (reply s))))))
```

D.6 Creating Instance Descriptions

An **a** or **an** expression represents the creation of an instance description. Its evaluation involves the evaluation of the concept and attribute fillers, followed by the creation of an instance description.

```
; expression "(an automobile (with color red))"
(define (new a-expression
             (with concept ≡c)
             (with attribute-sequence ≡as))
 (label self
  (create-unserialized
   (is-request (≡eval which-is (an expression-eval)) do
    (reply (new instance-description
                (with concept (ask c eval))
                (with attribute-sequence
                      (new eval-attribute-sequence
                           (with attribute-sequence as)
                           (with eval-message eval))))))
   ...)))
```

This abstraction evaluates a sequence of attributes from an a-expression, for creating an instance description.

```
(define (new eval-attribute-sequence
             (with attribute-sequence ≡as)
             (with eval-message ≡eval))
 (case-for as
  (is [] do (reply []))
  (is (a sequence (with first ≡f) (with rest ≡r)) do
    (reply (a sequence
              (with first (new eval-attribute
                               (with attribute f)
                               (with eval-message eval)))
              (with rest (new eval-attribute-sequence
                              (with attribute-sequence r)
                              (with eval-message eval))))))))
```

This abstraction evaluates a single attribute. An attribute is represented as a sequence of three elements: the kind, the relation, and the filler. It is evaluated by evaluating its filler.

```
(define (new eval-attribute
             (with attribute ≡a)
             (with eval-message ≡eval))
 (case-for a
  (is [≡kind ≡relation ≡filler] do
    (reply [kind relation (ask filler eval)]))))
```

D.7 Creating Actors

The **create** expression is for creating actors with specified behaviors. Here, we give a single implementation, of serializers. An implementation of create-unserialized would be almost identical. Expressions for creating actors whose internals are visible for pattern-matching also have a very similar implementation.

```
(define (new create-expression
             (with handler-groups ≡hgs))
 (create-unserialized
  (is-request (≡eval which-is
                (an expression-eval (with environment ≡e))) do
    (reply
      (new serializer
           (with environment e)
           (with descriptor
                 (case-for eval
                   (is (an expression-eval (with pattern ≡p)) do
                     (reply (ask p (a make-descriptor (with environment e)))))
                   (otherwise (is something do (reply (a something))))))
           (with state
                 (case-for eval
                   (is (an expression-eval (with state ≡s)) do
                     (reply s))
                   (otherwise (is something do (reply (a something))))))
           (with behavior
                 (new serializer-behavior
                      (with handler-groups
                            (new eval-handler-group-patterns
                                 (with handler-groups hgs)
                                 (with environment e))))))))
  ...))
```

This abstraction ripples down the groups of communication handlers in a **create** expression, calling on **eval-handler-sequence-patterns** to evaluate the patterns. The structure of a **create** expression is similar to the structure of **case-for** and **one-of** constructs. Communication handler groups are represented as a sequence of handler groups. A handler group is represented as a sequence of handlers. A handler is represented as a sequence containing three elements: a keyword indicating the significance of the pattern, a pattern, and a sequence of commands comprising the body.

```
(define (new eval-handler-group-patterns
             (with handler-groups ≡hgs)
             (with environment ≡e))
 (case-for hs
   (is [] do (reply []))
   (is (a sequence (with first ≡f) (with rest ≡r)) do
     (reply (a sequence
               (with first (new eval-handler-sequence-patterns
                           (with handler-sequence f)
                           (with environment e)))
               (with rest (new eval-handler-group-patterns
                              (with handler-groups r)
                              (with environment e))))))))
```

This abstraction ripples down a sequence of communication handlers, evaluating the patterns.

```
(define (new eval-handler-sequence-patterns
             (with handler-sequence ≡hs)
             (with environment ≡e))
 (case-for hs
   (is [] do (reply []))
   (is (a sequence
          (with first [≡keyword ≡pattern ≡body])
          (with rest ≡r))
     do
     (reply (new sequence
                 (with first [keyword (evaluate pattern e) body])
                 (with rest (new eval-handler-sequence-patterns
                                 (with handler-sequence r)
                                 (with environment e))))))))
```

D.8 Simple Commands

The **become** command designates a replacement for an actor, and causes the actor to become indistinguishable from its replacement.

```
; command: "(become ...)"
(define (new become-command (with replacement-actor ≡r))
 (create-unserialized
  (is-request (a command-eval
                 (with environment ≡e)
                 (with state ≡s))       do
    (reply (a become-effect
              (with replacement-actor
                    (ask r (an expression-eval
                              (with environment e)
                              (with default s)))))))
  ...))
```

The **send-to** command causes a communication to be sent to some target. The **reply-to** and **complain-to** commands are implemented in a very similar manner.

```
; command: "(send-to T C)", "(reply-to T M)", "(complain-to T M)"
(define (new send-to-command (with target ≡t) (with communication ≡c))
 (create-unserialized
  (is-request (a command-eval (with environment ≡e)) do
    (case-for (evaluate t e)
      (is ≡target do
        (case-for (evaluate c e)
          (is ≡com do
            (send-to target com)
            (reply (a completed-command-effect)))
          (complaint ≡m do
            (complain-to target m)
            (reply (a completed-command-effect)))))))
  ...))
```

The **send** command causes a reply to be sent to some default customer or a complaint to be sent to some default complaint-department. The **reply** and **complain** commands are implemented in a similar manner.

```
; command: "(send ...)", "(reply ...)", "(complain ...)"
(define (new send-command (with communication ≡com))
 (create-unserialized
  (is-request (a command-eval
                (with environment ≡e)
                (with communication ≡c)) do
    (case-for c
      (is (a request
            (with customer ≡cus)
            (with complaint-department ≡cd)) do
        (case-for (evaluate com e)
          (is (≡r which-is (a reply)) do
            (reply (a send-effect
                     (with communication r)
                     (with target cus))))
          (is (≡r which-is (a complaint)) do
            (reply (a send-effect
                     (with communication r)
                     (with target cd))))))))
  ...))
```

D.9 Composite Constructs

D.9.1 Case-for Construct

The case-for construct dispatches on the response from the evaluation of an expression. It can have several groups of handlers for the response. They are represented as a sequence of sequences of handlers. A handler is represented as a sequence with three elements: a keyword [`is` or `complaint`], a pattern, and a command sequence comprising the body.

```
(define (new case-for-construct (with test-expression ≡te)
            (with handler-groups ≡hgs))
  (create-unserialized
   (is-request (a command-eval
                  (with environment ≡e)
                  (with communication ≡c)
                  (with state ≡s)) do
     (case-for (new find-case-for-body-from-expr (with test-expression te)
                                                 (with handler-groups hgs)
                                                 (with environment e))
       (is (a found-case-for-body (with body ≡b) (with environment ≡e)) do
         (reply (new eval-command-sequence (with command-sequence b)
                                           (with environment e)
                                           (with communication c)
                                           (with state s))))
       (is (a could-not-find) do
         (case-for c
           (is (a complaint) do (send c))
           (is (a reply) do
             (complain (an unmatched-reply (with reply c))))))))
   (is-communication (≡com which-is
                      (a request
                         (with message
                               (an expression-eval
                                   (with environment ≡e))))) do
     (case-for (new find-case-for-body-from-expr (with test-expression te)
                                                 (with handler-groups hgs)
                                                 (with environment e))
       (is (a found-case-for-body (with body ≡b) (with environment ≡e)) do
         (reply (new eval-expression-body
                     (with command-sequence b)
                     (with environment e)
                     (with communication com)
                     (with state (ask com (a descriptor))))))
       (complaint (a could-not-find (with communication ≡c)) do
         (case-for c
           (is (a complaint) do (send c))
           (is (a reply) do
             (complain (an ummatched-reply (with reply c))))))))
   ...))
```

This abstraction evaluates the test expression in the **case-for** construct, then sets up **find-case-for-body** to do the rest of the work.

```
(define (new find-case-for-body-from-expr
             (with test-expression ≡te)
             (with handler-groups ≡hgs)
             (with environment ≡env))
 (case-for (evaluate te env)
   (is ≡v do
     (reply (new find-case-for-body
                 (with keyword 'is)
                 (with message v)
                 (with handler-groups hgs)
                 (with environment env))))
   (complaint ≡m do
     (reply (new find-case-for-body
                 (with keyword 'complaint)
                 (with message m)
                 (with handler-groups hgs)
                 (with environment env))))))
```

This abstraction ripples down the sequence of handler groups, sequentially trying each for a match. Each handler group corresponds to a set of handlers nested in an **otherwise** clause.

```
; for each handler in 'hs: eval pattern, try to match, return body and env.
(define (new find-case-for-body
             (with keyword ≡k)
             (with message ≡m)
             (with handler-groups ≡hgs)
             (with environment ≡env))
 (case-for hgs
  (is [] do (reply (a could-not-find)))
  (is (a sequence (with first ≡f) (with rest ≡r)) do
    (case-for (new find-case-for-body-1
                   (with keyword k)
                   (with message m)
                   (with handler-sequence f)
                   (with environment env))
      (is ≡x do (reply x))
      (is (a could-not-find) do
        (reply (new find-case-for-body
                    (with keyword k)
                    (with message m)
                    (with handler-groups r)
                    (with environment env))))))))
```

This abstraction ripples down a handler group [a sequence of handlers], looking for a matching handler. Specifications state that these patterns are checked concurrently. This can be achieved in a concrete implementation by eagerly evaluating recursive calls of this abstraction.

```
(define (new find-case-for-body-1
            (with keyword ≡k)
            (with message ≡m)
            (with handler-sequence ≡hs)
            (with environment ≡env))
 (case-for hs
  (is [] do (reply (a could-not-find)))
  (is (a sequence
         (with first [≡keyword ≡pattern ≡body])
         (with rest ≡r))                                    do
    (if (= keyword k)
      (then do
        (case-for (ask (evaluate-pattern env)
                       (a match
                          (with bindings (new empty-layer))
                          (with object m)))
          (is (a successful-match (with bindings ≡bin)) do
            (reply (a found-case-for-body
                      (with body bod)
                      (with environment
                            (new environment
                                 (with primary bin)
                                 (with secondary env))))))
          (is (a failed-match) do
            (reply (new find-case-for-body-1
                        (with keyword k)
                        (with message m)
                        (with handler-sequence r)
                        (with environment env))))))
      (else do
        (reply (new find-case-for-body-1
                    (with keyword k)
                    (with message m)
                    (with handler-sequence r)
                    (with environment env)))))
```

D.9.2 One-of Construct

The **one-of** construct is a very flexible and general construct for making decisions based on Boolean predicates. It is similar in structure to the **case-for** and **create** expressions. It is represented as a sequence of arm groups, each of which is a sequence of arms. The arm groups are tried sequentially, looking for one whose predicate yields truth. The arms in each arm group are specified as being tried concurrently. Each arm consists of a predicate and a command sequence which serves as the body.

```
(define (new one-of-construct (with arm-groups ≡ags))
 (label self
  (create-unserialized
   (is-request (a command-eval
                (with environment ≡e)
                (with communication ≡c)
                (with state ≡s))          do
     (reply (new eval-command-sequence
                 (with command-sequence
                       (new find-one-of-body (with arm-groups ags)
                                             (with environment e)))
                 (with environment e)
                 (with communication c)
                 (with state s))))
   (is-communication
     (≡com which-is
      (a request (with message
                       (an expression-eval (with environment ≡e)))))
     do
     (reply (new eval-expression-body
                 (with command-sequence
                       (new find-one-of-body
                            (with arm-groups ags)
                            (with environment e)))
                 (with environment e)
                 (with communication com)
                 (with state (ask com (a descriptor))))))
   ...)))
```

This abstraction ripples sequentially down the sequence of handler groups, calling **find-one-of-body-1** on each group, until a body is found.

```
(define (new find-one-of-body
             (with arm-groups ≡ags)
             (with environment ≡env))
 (case-for as
   (is [] do (complain (a not-found)))
   (is (a sequence (with first ≡f) (with rest ≡r)) do
     (case-for (new find-one-of-body-1
                    (with arm-sequence f)
                    (with environment env))
       (is ≡b do (reply b))
       (complaint (a not-found) do
         (reply (new find-one-of-body
                     (with arm-groups r)
                     (with environment env))))))))
```

This abstraction ripples down an arm group, which is represented as a sequence of arms, looking for an arm whose predicate yields truth. An arm is represented by a sequence with two elements: a predicate and a command

sequence.

```
(define (new find-one-of-body-1
             (with arm-sequence ≡as)
             (with environment ≡env))
 (case-for as
   (is [] do (complain (a not-found)))
   (is (a sequence
          (with first [≡predicate ≡body])
          (with rest ≡r)) do
     (if (evaluate predicate env)
       (then do (reply body))
       (else do (reply (new find-one-of-body-1
                            (with arm-sequence r)
                            (with environment env))))))))
```

D.9.3 Let Construct

The **let** construct provides a general way to perform a number of pattern-matches, then evaluate some commands using any bindings which resulted in the matches. A degenerate, but very useful, case of this is simply binding an identifier to the result of an expression. The group of matchers in a **let** construct are represented as a sequence of matchers. Each matcher is represented as a sequence containing two elements: a pattern for the match and an object for the match.

```
(define (new let-construct
             (with matcher-sequence ≡ms)
             (with body ≡b))
 (label self
  (create-unserialized
   (is-request (a command-eval
                (with environment ≡e)
                (with communication ≡c)
                (with state ≡s))        do
     (reply (new eval-command-sequence
                 (with command-sequence b)
                 (with environment
                       (new environment
                            (with primary
                                  (new process-matcher-sequence
                                       (with matcher-sequence ms)
                                       (with bindings (new empty-layer))
                                       (with environment e)))
                            (with secondary e)))
                 (with communication c)
                 (with state s))))))
   (is-communication
     (≡com which-is
      (a request (with message
                       (an expression-eval (with environment ≡e)))))
     do
     (reply (new eval-expression-body
                 (with command-sequence b)
                 (with environment
                       (with primary
                             (new process-matcher-sequence
                                  (with matcher-sequence ms)
                                  (with bindings (new empty-layer))
                                  (with environment e)))
                       (with secondary e))
                 (with communication com)
                 (with state (ask com (a descriptor))))))))
   ...)))
```

This abstraction ripples down the sequence of matchers, performing each match. If successful, it replies with a layer of bindings established during the matching.

```
(define (new process-matcher-sequence
            (with matcher-sequence ≡ms)
            (with bindings ≡b)
            (with environment ≡env))
 (case-for ms
  (is [] do (reply b)))
  (is (a sequence
         (with first [≡pattern ≡object])
         (with rest ≡r)) do
    (case-for (ask (evaluate pattern env)
                   (a match
                      (with bindings b)
                      (with object (evaluate object env))))
      (is (a failed-match) do (complain (a cannot-match)))
      (is (a successful-match (with bindings ≡b)) do
        (reply (new process-matcher-sequence
                    (with matcher-sequence r)
                    (with bindings b)
                    (with environment env)))))))
```

D.9.4 Other Constructs

The **if** construct can be implemented as a syntactic extension of Act2, or can be implemented in a way similar to the implementation of **one-of**. An **if** construct of the form

```
(if predicate
  (then do then-commands)
  (else do else-commands))
```

has the same meaning as a **one-of** construct of the form

```
(one-of
  (if predicate do then-commands)
  (otherwise (if true do else-commands)))
```

The **label** expression can also be implemented either directly, or as a syntactic extension. A **label** expression of the form `(label` *symbol expression*`)` has the same meaning as a **let** expression of the form

```
(let ((≡symbol match (delay expression))) do
  (reply symbol))
```

Notice the presence of the **delay** expression, to postpone the evaluation of the expression. This is necessary, since the expression can refer to its own value.

D.10 Subsidiary Abstractions

D.10.1 Environments and Layers

Environments are composed of layers. The top layer of each environment can be grown with new bindings of symbols to values. Therefore, environments are serialized. Layers are unserialized, for speed. The top layer of an environment is called its *primary layer.* Each environment also has a *secondary environment.*

```
(define (new environment
             (with primary ≡pe)
             (with secondary ≡se))
  (create
   (is-request (a grow (with symbol ≡s) (with value ≡v)) do
     (let ((≡x match (new environment
                          (with primary
                                (ask pe (a grow
                                           (with symbol s)
                                           (with value v))))
                          (with secondary ≡se))))
       do (become x) (reply x)))
   (is-request (a lookup (with symbol ≡s)) do
     (case-for (ask pe (a lookup (with symbol s)))
       (is ≡value do (reply value))
       (complaint ≡message do
         (reply (ask se (a lookup (with symbol s)))))))
   (is-request (≡msg which-is (a present (with symbol ≡s))) do
     (reply (or (ask pe msg) (ask se msg))))
   ...))
```

In order to implement layers, we need to implement empty layers which accept the same communications as layers and environments. An empty layer has no bindings, and replies with a layer when asked to grow.

```
(define (new empty-layer)
 (label self
  (create-unserialized
    (is-request (a grow (with symbol ≡s) (with value ≡v)) do
      (reply (new layer
                  (with symbol s)
                  (with value v)
                  (with next self))))
    (is-request (a lookup (with symbol ≡s)) do
      (complain (a missing-binding (with symbol s))))
    (is-request (a present) do (reply false))
    ...)))
```

A layer is an unserialized collection of bindings of symbols to values. It can be implemented as a recursive data structure, as shown here.

```
(define (new layer
             (with symbol ≡s)
             (with value ≡v)
             (with next ≡n))
 (label self
  (create-unserialized
    (is-request (≡msg which-is (a lookup (with symbol ≡sym))) do
      (if (= s sym)
        (then do (reply v))
        (else do (reply (ask n msg)))))
    (is-request (≡msg which-is (a present (with symbol ≡sym))) do
      (if (= s sym)
        (then do (reply true))
        (else do (reply (ask n msg)))))
    (is-request (a grow (with symbol ≡sym) (with value ≡val)) do
      (reply (new layer
                  (with symbol sym)
                  (with value val)
                  (with next self))))
    ...)))
```

D.10.2 Atomic Descriptions

The friendly interface to atomic descriptions is through the **concept** abstraction. It allows a programmer to create an atomic description by providing only its name.

```
(define (new concept (with name ≡n))
 (reply
  (new atomic-description
       (with name n)
       (with encryption-id (new encryption-id))
       (with description-stuff (new description-stuff))
       (with implementation-stuff (new implementation-stuff))
       (with creation-stuff (new creation-stuff)))))
```

The full detail of atomic descriptions is managed by the **atomic-description** abstraction. Each atomic description has a name, which is a symbol used mostly for identification by humans, as well as an encryption-id, which is a unique discriminator used to distinguish between independently-created atomic descriptions. In addition, atomic descriptions have room for installing description-

lattice information (description-stuff), for installing implementation information (implementation-stuff), and for aiding in bottoming out instance descriptions (creation-stuff). The details of the latter are not important for this level of description. The atomic description itself is unserialized, but has some serialized acquaintances, such as **implementation-stuff.**

```
(define (new atomic-description
             (with name ≡nam)
             (with encryption-id ≡eid)
             (with description-stuff ≡des)
             (with implementation-stuff ≡imp)
             (with creation-stuff ≡cre))
 (label self
  (create-unserialized
   (is-request (a match (with object ≡o) (with bindings ≡b)) do
     ;; eventually match will be done with a low-level comparison of 'eid.
     ;; 'eid is a unique encryption id associated with an atomic
     ;;description when it is created with a (defconcept).
     (one-of
      (if (identical self o) do
        (reply (a successful-match (with bindings b))))
      ;; should eventually let description-system have a crack at it.
      (otherwise do (reply (a failed-match)))))
   (is-request (a converse-match ...) do ...)
   (is-request (≡msg which-is
                (an install-implementation
                    (with environment ≡e)
                    (with creation-pattern ≡cp)
                    (with creation-expression ≡ce)))  do
     (reply (ask imp msg)))
   (is-request (≡msg which-is (a summarize-implementation)) do
     (reply (ask imp msg)))
   ...))
```

When a **define** is evaluated, implementation information usually gets installed in an atomic description as a result. It is in the implementation acquaintance that this information is installed. This actor must be serialized, so redefinitions can occur.

```
(define (new implementation
            (with expression ≡exp)
            (with environment ≡env)
            (with pattern ≡pat))
  (create
    (is-request (an install-implementation
                    (with expression ≡exp1)
                    (with environment ≡env1)
                    (with pattern ≡pat1))    do
      (become (new implementation
                  (with expression exp1)
                  (with environment env1)
                  (with pattern pat1)))
      (reply (a completion-report)))
    (is-request (a summarize-implementation) do
      (reply (an installation
                 (with expression exp)
                 (with environment env)
                 (with pattern pat))))
    ...))
```

The **implementation-stuff** abstraction is a trivial interface to the **implementation** abstraction. It is used to create an initial, null implementation. This implementation simply complains that the abstraction does not yet have an implementation.

```
(define (new implementation-stuff)
  (new implementation
       (with expression
             (function
               (let ((something match 1)) do
                 (complain (an unimplemented-abstraction)))))
       (with environment (new empty-layer))
       (with pattern (a something))))
```

D.10.3 Instance Descriptions

The **instance-description** abstraction implements instance descriptions for Act2. An instance description is represented as a concept and a sequence of attributes. Each attribute is represented as a sequence containing three elements: the attribute kind, the attribute relation, and the attribute filler.

```
(define (new instance-description
            (with concept ≡c)
            (with attribute-sequence ≡as))
 (label self
  (create-unserialized
   (is-request (a match (with bindings ≡b) (with object ≡o)) do
     (reply (ask o (a converse-match
                      (with pattern (an instance-description
                                        (with concept c)
                                        (with attributes as)))
                      (with bindings b)))))
   (is-request (a converse-match
                  (with pattern
                        (an instance-description
                            (with concept ≡c1)
                            (with attributes ≡as1)))
                  (with bindings ≡b))
     do ;; try to match self with instance-pattern.
     (if (identical c c1)
       (then do
         (reply (new match-attributes
                     (with patterns as1)
                     (with objects as)
                     (with bindings b))))
       (else do (reply (a failed-match)))))
   (is-request (a make-descriptor (with environment ≡e)) do
     (reply (new instance-description
                 (with concept c)
                 (with attribute-sequence []))))
   ...)))
```

This abstraction matches the attribute-sequences from two instance descriptions.

```
(define (new match-atributes
            (with patterns ≡ps)
            (with objects ≡os)
            (with bindings ≡b))
 (case-for ps
   (is [] do (reply (a successful-match (with bindings b))))
   (is (a sequence
          (with first [≡kind ≡relation ≡filler])
          (with rest ≡res))
     do ;; try to find a matching attribute in os.
     (case-for (new match-attribute-in-sequence
                    (with kind kind)
                    (with relation relation)
                    (with filler filler)
                    (with objects os)
                    (with bindings b))
       (is (a successful-match (with bindings ≡b)) do
         (reply (new match-attributes
                     (with patterns res)
                     (with objects os)
                     (with bindings b))))
       (otherwise (is something do (reply (a failed-match))))))))
```

This abstraction attempts to match a dismantled pattern attribute to an attribute in a sequence of object attributes.

```
(define (new match-attribute-in-sequence
            (with kind ≡k)
            (with relation ≡r)
            (with filler ≡f)
            (with objects ≡os)
            (with bindings ≡b))
 (case-for os
   (is [] do (reply (a failed-match)))
   (is (a sequence (with first [≡k1 ≡r1 ≡f1])
                   (with rest ≡res))
     do ;; try to find a matching attribute in os.
     (one-of
      (if (identical r r1) do ;; found relation, now try to match filler.
        (reply (ask f (a match (with bindings b) (with object f1)))))
      (otherwise (if true do ;; keep looking for relation.
        (reply (new match-attribute-in-sequence
                    (with kind k)
                    (with relation r)
                    (with filler f)
                    (with objects res)
                    (with bindings b))))))))))
```

D.10.4 Serializers

This abstraction represents serialized actors. The implementation of unserialized actors resembles this.

```
(define (new serializer
             (with descriptor ≡d)
             (with state ≡s)
             (with behavior ≡b)
             (with environment ≡ce))
 (label self
  (create
   (is-communication ≡com do
     (send-to b (a process-communication
                  ;; the incoming communication:
                  (with communication com)
                  ;; the creation environment:
                  (with environment ce)
                  ;; the actor's "type":
                  (with descriptor d)
                  ;; a description of the actor, including internals.
                  (with state s)
                  ;; the actor itself:
                  (with self self)))))))
```

This abstraction represents a serializer's behavior, or script.

```
(define (new serializer-behavior
             (with handler-groups ≡hgs))
  (create
   (is-request (a process-communication
                 (with communication ≡com)
                 (with environment ≡ce)
                 (with descriptor ≡d)
                 (with state ≡s)
                 (with self ≡self))
     do
     (case-for (new eval-matching-handler
                  (with handler-groups hgs)
                  (with descriptor d)
                  (with state s)
                  (with environment ce)
                  (with communication com)
                  (with self self))
       (is (a become-effect (with replacement-actor ≡ra)) do (become ra))
       (otherwise
        (is something do )
        (complaint (a no-match) do
          (complain (a rejected-communication (with communication com)))))))))
```

When an actor accepts a communication, it calls this abstraction to find a

handler for the communication and evaluate the corresponding body.

```
(define (new eval-matching-handler
             (with handler-groups ≡hgs)
             (with descriptor ≡d)
             (with state ≡s)
             (with environment ≡ce)
             (with communication ≡com)
             (with self ≡me))
 (case-for hgs
   (is [] do
     (reply (new try-default-handlers
                 (with descriptor d)
                 (with communication com)
                 (with state s)
                 (with self me))))
   (is (a sequence (with first ≡f) (with rest ≡r)) do
     (case-for (new find-and-eval-winning-handler
                    (with handler-sequence f)
                    (with environment ce)
                    (with communication com)
                    (with state s))
       (is ≡x do (reply x))
       (complaint (a no-match) do
         (case-for (new try-default-handlers
                        (with descriptor d)
                        (with communication com)
                        (with state s)
                        (with self me))
           (is ≡x do (reply x))
           (complaint (a no-match) do
             (reply (new eval-matching-handler-without-defaults
                         (with handler-groups r)
                         (with descriptor d)
                         (with state s)
                         (with environment ce)
                         (with communication com)
                         (with self me)))))))))))
```

This abstraction implements default handlers for all actors. These handle communications such as requests to match, converse-match, and print.

```
(define (new try-default-handlers
             (with descriptor ≡d)
             (with communication ≡c)
             (with state ≡s)
             (with self ≡me))
 ;; should be able to write these as user code.
 (case-for c
   (is (a request (with message (a match
                                  (with object ≡o)
                                  (with bindings ≡b)))
                  (with customer ≡cus))
     do ;; need to be same actor as me for a match.
     (one-of
       (if (identical me o) do
         (reply-to cus (a successful-match (with bindings b))))
       (otherwise do (reply-to cus (a failed-match)))))
   (is (a request
         (with message (which-is ≡m (a converse-match)))
         (with customer ≡cus))
     do ;; This match is really a type-check, so let descriptor try.
     (reply-to cus (ask d m)))
   (otherwise (is something do (complain (a no-match))))))
```

This abstraction is similar to **eval-matching-handler**, but is for those handler groups appearing after the first. The difference is that this abstraction will not try the default handlers again.

```
(define (new eval-matching-handler-without-defaults
             (with handler-groups ≡hgs)
             (with descriptor ≡d)
             (with state ≡s)
             (with environment ≡ce)
             (with communication ≡com)
             (with self ≡me))
 (case-for hgs
   (is [] do (complain (a no-match)))
   (is (a sequence (with first ≡f) (with rest ≡r)) do
     (case-for (new find-and-eval-winning-handler
                    (with handler-sequence f)
                    (with environment ce)
                    (with communication com)
                    (with state s))
       (is ≡x do (reply x))
       (complaint (a no-match) do
         (reply (new eval-matching-handler-without-defaults
                     (with handler-groups r)
                     (with descriptor d)
                     (with state s)
                     (with creation-environment ce)
                     (with communication com)
                     (with self me))))))))
```

This abstraction attempts to find and evaluate a handler for the incoming communication. It looks for it in a single handler group, which is represented as a sequence of handlers.

```
(define (new find-and-eval-winning-handler
             (with handler-sequence ≡hs)
             (with environment ≡ce)
             (with communication ≡c)
             (with state ≡s))
 (case-for hs
   (is [] do (complain (a no-match)))
   (is (a sequence (with first ≡f) (with rest ≡r)) do
     (case-for (new match-handler (with handler f) (with communication c))
       (is (a successful-match
              (with bindings ≡bin)
              (with body ≡bod))     do
         (reply (new eval-handler-body
                     (with state s)
                     (with body bod)
                     (with communication c)
                     (with environment
                           (new environment (with primary bin)
                                            (with secondary ce))))))
       (is (a failed-match) do
         (reply (new find-and-eval-winning-handler
                     (with handler-sequence r)
                     (with environment ce)
                     (with communication c)
                     (with state s))))))))
```

This abstraction attempts to match a communication handler with an incoming communication. The representation of a communication handler is a sequence with three elements, a keyword, a pattern, and a body.

```
(define (new match-handler
            (with handler ≡han)
            (with communication ≡com))
 (case-for han
   (is [≡key ≡pat ≡bod] do
     (case-for key
       (is 'is-communication do
         (case-for (ask pat (a match
                              (with bindings (new empty-layer))
                              (with object com)))
           (is (a successful-match (with bindings ≡bin)) do
             (reply (a successful-match
                       (with bindings bin)
                       (with body bod))))
           (is (a failed-match) do (complain (a failed-match)))))
       (is 'is-request do
         (case-for com
           (is (a request (with message ≡msg)) do
             (case-for (ask pat (a match
                                  (with bindings (new empty-layer))
                                  (with object msg)))
               (is (a successful-match (with bindings ≡bin)) do
                 (reply (a successful-match
                           (with bindings bin)
                           (with body bod))))
               (is (a failed-match) do (complain (a failed-match)))))
           (otherwise (is something do (complain (a failed-match))))))
       (is 'is-reply do
         (case-for com
           (is (a reply (with message ≡msg)) do
             (case-for (ask pat (a match
                                  (with bindings (new empty-layer))
                                  (with object msg)))
               (is (a successful-match (with bindings ≡bin)) do
                 (reply (a successful-match
                           (with bindings bin)
                           (with body bod))))
               (is (a failed-match) do (complain (a failed-match)))))
           (otherwise (is something do (complain (a failed-match))))))
       (is 'is-complaint do
         (case-for com
           (is (a complaint (with message ≡msg)) do
             (case-for (ask pat (a match
                                  (with bindings (new empty-layer))
                                  (with object msg)))
               (is (a successful-match (with bindings ≡bin)) do
                 (reply (a successful-match
                           (with bindings bin)
                           (with body bod))))
               (is (a failed-match) do (complain (a failed-match)))))
           (otherwise (is something do (complain (a failed-match))))))
```

D.10.5 Evaluating Composite Expression Bodies

When composite constructs such as **case-for**, **one-of**, and **let** are used as expressions, the commands in their bodies are evaluated by the **eval-expression-body** abstraction. It calls **eval-command-sequence** to evaluate the body, then calls **process-expression-effects** to condense the result into a single **send-effect**.

```
(define (new eval-expression-body
             (with command-sequence ≡cs)
             (with environment ≡e)
             (with communication ≡c)
             (with state ≡s))
  (case-for (new process-expression-effects
                 (with environment e)
                 (with effects
                       (new eval-command-sequence
                            (with command-sequence cs)
                            (with environment e)
                            (with communication c)
                            (with state s))))
    (is (a send-effect (with communication com)) do (send com))
    (otherwise (is something do (complain (a failure))))))
```

This abstraction processes a sequence of effects from the evaluation of a body of commands. It assumes the context is for a construct which has been used as an expression. There should be no **become-effect**. There should be exactly one **send-effect**.

```
(define (new process-expression-effects
             (with effect-sequence ≡es)
             (with environment ≡env))
 (case-for es
   (is (a send-effect (with communication ≡c)) do (reply es))
   (is (a completed-command-effect) do (reply es))
   (is (a become-effect) do (complain es))
   (is (a send-effect) do (complain es))
   (is [] do (reply []))
   (is (a sequence (with first ≡f) (with rest ≡r)) do
     (case-for (new process-expression-effects (with effect-sequence f)
                                               (with environment env))
       (is (a send-effect (with communication ≡c)) do
         (case-for (new process-expression-effects (with effect-sequence r)
                                                   (with environment env))
           (is (a send-effect) do (complain (a failure)))
           (otherwise (is something do (reply (a send-effect
                                                (with communication c)))))))
       (otherwise (is something do (reply (new process-expression-effects
                                               (with effect-sequence r)
                                               (with environment env))))))))
```

D.10.6 Evaluating Communication Handler Bodies

The process of evaluating a communication handler body consists of evaluating its body. This transforms a sequence of commands into a sequence of effects, which we the process with **process-handler-effects.**

```
(define (new eval-handler-body
             (with state ≡s)
             (with communication ≡c)
             (with environment ≡e)
             (with body ≡b))
  (reply (new process-handler-effects
              (with effect-sequence
                    (new eval-command-sequence
                         (with command-sequence b)
                         (with environment e)
                         (with communication c)
                         (with state s))))))
```

This abstraction processes a sequence of effects from the evaluation of a communication handler body. No more than one **become-effect** should be encountered.

```
; 'es might be just an effect, instead of a sequence of effects.
(define (new process-handler-effects
             (with effect-sequence ≡es))
 (case-for es
  (is (a completed-command-effect) do (reply es))
  (is (a send-effect (with communication ≡com) (with target ≡tar)) do
    (send-to tar com)
    (reply (a completed-command-effect)))
  (is (a become-effect (with replacement-actor ≡ra)) do (reply es))
  (is [] do (reply (a completed-command-effect)))
  (is (a sequence (with first ≡f) (with rest ≡r)) do
    (case-for (new process-handler-effects
                   (with effect-sequence f))
      (is (a become-effect (with replacement-actor ≡ra)) do
        (case-for (new process-handler-effects
                       (with effect-sequence r))
          (is (a become-effect) do (complain (a failure)))
          (is (a completed-command-effect) do
            (reply (a become-effect (with replacement-actor ra))))))
      (is (a completed-command-effect) do
        (reply (new process-handler-effects
                    (with effect-sequence r))))))))
```

D.10.7 Evaluating a Command Sequence

The evaluation of a sequence of commands produces a sequence of effects.

```
(define (new eval-command-sequence
             (with command-sequence ≡cs)
             (with environment ≡env)
             (with communication ≡com)
             (with state ≡s))
 (case-for cs
  (is [] do (reply []))
  (is (a sequence (with first ≡f) (with rest ≡r)) do
    (reply (new sequence
                (with first (a command-eval
                               (with environment env)
                               (with communication com)
                               (with state s)))
                (with rest (new eval-command-sequence
                                (with command-sequence r)
                                (with environment env)
                                (with communication com)
                                (with state s))))))))
```

Appendix E

Pre-Defined Names, Actors, and Protocols

When a user first encounters an Act2 listener, there will already be an environment associated with the listener. This environment will contain mappings from a number of standard identifiers to useful actors. This initial community of actors serves as a foundation upon which one can build useful actor communities of his own. The following table describes the actors in this standard initial environment. It may not be wise to rebind some of these names in your environment, or in computations.

true An actor which behaves like a logical value of truth.

false An actor which behaves like a logical value of falsity.

standard-act2-evaluation-environment
This is the environment currently associated with the listener. The **defname** expression extends this environment.

standard-act2-expressions
An environment used for parsing. It is a mapping from symbols which serve as expression keywords to parsers which can create abstract syntax actors from a list-structure representation of the expression. The **defexpression** expression extends this environment.

standard-act2-commands
An environment used for parsing. It is a mapping from symbols which serve as command keywords to parsers which can create abstract syntax actors from a list-structure representation of the command. The **defcommand** expression extends this environment.

standard-act2-initial-evaluation-environment

An environment containing the pre-defined symbols for Act2. This environment may be shared by other users and should not be extended. It serves as a secondary environment for ***standard-act2-evaluation-environment***, which can be extended at will.

standard-act2-initial-expressions

An environment containing standard expression keyword/parser mappings. It serves as a secondary environment for ***standard-act2-expressions***, and might be shared among different users. Customizations should be installed by extending ***standard-act2-expressions***, and ***standard-act2-initial-expressions*** should only be used for reference.

standard-act2-initial-commands

An environment containing standard command keyword/parser mappings. It serves as a secondary environment for ***standard-act2-commands***, and might be shared among different users. Customizations should be installed by extending ***standard-act2-commands***, and ***standard-act2-initial-commands*** should only be used for reference.

In addition, a number of identifiers are bound to atomic descriptions for the concepts of each of the instance descriptions used as messages in the standard communication protocols described below. These include: abs, add1, are-you, attribute, become-effect, command-compile, command-eval, communication, compile, complaint, completed-command-effect, concept, concept-for-instance-description, converse-match, could-not-find, creation-info, evenp, expression-compile, expression-eval, failed-match, failure, found-case-for-body, grow, if, install-implementation, installation, lookup, make-descriptor, match, match-compile, merge-attributes, minus, minusp, name, oddp, plusp, present, process-communication, ready-effect, reply, request, requisition, send-effect, sequence, something, successful-match, summarize-implementation, zerop, =, >, >=, ≥, <, <=, ≤, +, -, *, //, and ↑.

E.1 Common Protocol for All Actors

All actors Act2 deals with should handle requests with the following messages. Actors created with any variation of the Act2 **create** expression are provided with handlers for these communications, by default.

`(a = (with operand ...))`
: Reply with a truth value indicating whether or not you and the operand are "the same actor". The exact behavior expected depends upon whether or not the actors are serialized, and upon the sophistication of the actor making the comparison.

`(an are-you (with what ...))`
: Reply with a truth value indicating whether or not you know yourself to be an instance of the specified concept. The `what` is often a symbol, as in `(an are-you (with what 'sequence))`.

`(a match (with object ...) (with bindings ...))`
: Reply either with `(a failed-match)` or `(a successful-match (with bindings ...))`, depending upon whether you as a pattern match the `object`, given the specified symbol-to-actor `bindings`. A successful match might involve extending the set of `bindings`.

`(a converse-match (with pattern ...) (with bindings ...))`
: Reply either with `(a failed-match)` or `(a successful-match (with bindings ...))`, depending upon whether the `pattern` matches you, given the `bindings`.

E.2 Surface Syntax Actors

Act2 expressions are read in by an Act2 listener as list structure, symbols, and numbers. Immediately after reading in such a surface syntax actor, the listener asks it to parse itself as an expression. It may, in turn, ask surface syntax actors within itself to parse themselves either as expressions or as commands. The result of this parsing process is expected to be an abstract syntax actor.

```
(a parse-yourself-as-expression
        (with expression-keywords ...)
        (with command-keywords ...))
```
Reply with an abstract syntax actor representing the expression you denote, otherwise complain. Lists can represent a variety of expressions, so they make use of the `expression-keywords` environment. The list will scan itself from left to right, looking for a symbol which serves as a keyword. Keywords are symbols which are bound to parsers in the **expression-keywords** environment. When it has found a parser, the list asks it to parse the list into an abstract syntax actor using the keyword environments. Notice that the language can be extended syntactically simply by adding new bindings to these keyword environments.

```
(a parse-yourself-as-command
        (with expression-keywords ...)
        (with command-keywords ...))
```
Reply with an abstract syntax actor representing the command you denote, otherwise complain. The parsing process is very similar to that for expressions.

E.3 Parsers

Parsers in Act2 are used to help parse list structure, as indicated immediately above. They accept a list which represents Act2 `source` code, as well as environments in which keywords are bound to parsers. Parsing a list will typically involve asking elements in the list to parse themselves either as expressions or as commands.

```
(an expression-parse
        (with source ...)
        (with expression-keywords ...)
        (with command-keywords ...))
```
If possible, parse the list structure presented as the `source` into an abstract syntax actor representing one of the particular kinds of

expressions you were created to parse. If you cannot make sense of the list structure, complain with a revealing message.

(a command-parse
(with source ...)
(with expression-keywords ...)
(with command-keywords ...))
Try to parse the **source** list structure into an abstract syntax actor representing a command.

E.4 Abstract Syntax Actors

Abstract syntax actors represent Act2 expressions and/or commands. When they are asked to evaluate themselves as such, they perform the actions characteristic of the constructs they represent.

(an expression-eval (with environment ...))
Evaluate yourself as an expression, resolving names in the environment provided. Respond with a reply containing the expression value or with a complaint explaining the reason you cannot successfully produce such a value.

(a command-eval
(with environment ...)
(with communication ...)
(with state ...))
Evaluate yourself as a command, resolving names in the environment provided. You may use the extra context information provided in the communication being processed, or in a description of the actor in whose communication handlers you appear. Respond with an appropriate effect, such as **(a completed-command-effect)**, **(a send-effect (with communication ...))**, or **(a become-effect (with replacement ...))**, indicating what's been done or what remains to be done.

E.5 Environments and Layers

Environments and layers speak with the following protocol.

`(a lookup (with symbol ...))`
: If you contain a binding of the `symbol` to some actor, reply with that actor; if not, complain.

`(a present (with symbol ...))`
: Reply with a truth value indicating whether or not you contain a binding of the `symbol` to some actor.

`(a grow (with symbol ...) (with value ...))`
: Extend yourself with a binding of the `symbol` to the **value**, then reply with the resulting environment or layer. Environments are serialized and reply with (a changed version of) themselves. Layers are unserialized and reply with a new layer.

E.6 Rock-Bottom Numbers

Rock-bottom numbers obey the common protocols, the protocols for surface syntax actors and for abstract syntax actors, as well as the protocols below.

`(a + (with operand ...))`
: Reply with the number which is the sum of yourself and the `operand`. Other arithmetic operations understood are: subtraction (-), multiplication (*), division (//, ÷), exponentiation (↑), maximization (max), minimization (min).

`(a < (with operand ...))`
: Reply with a truth value indicating whether or not you consider yourself "less than" the `operand`. Other relational operations understood are: greater than (>), equality (=), greater than or equal (>=, ≥), less than or equal (<=, ≤).

`(an are-you (with what ...))`
: Rock-bottom numbers understand the concepts: `'number`, `'integer`, `'real`, `'whole-number`, and `'natural-number`.

E.7 Symbols

Primitive symbols obey the common protocols, the protocols for surface syntax actors and for abstract syntax actors, as well as the protocols below. The special symbol, `T`, also behaves as the logical truth value representing validity. The special symbol, `NIL`, also behaves as the logical truth value representing falsity, and as an empty list or sequence.

E.8 Sequences and Lists

Sequences and lists represent linearly-ordered collections of actors. They obey the common protocols, the protocols for surface syntax actors and for abstract syntax actors, as well as the protocols below.

`(a first)` Reply with the first element of the list or sequence. Complain if you are an empty list.

`(a rest)` Reply with the list or sequence consisting of all elements except the first. Complain if you are an empty list.

`(an nth (with number ...))`
Reply with the `number`th element in the list or sequence. Complain if there is no such element.

E.9 Atomic Descriptions

Atomic descriptions serve as concepts for instance descriptions, and as an organizational tool for the Act2 implementation. They obey the common protocols, as well as the protocols below.

```
(an install-implementation
            (with environment ...)
            (with creation-pattern ...)
```

`(with creation-expression ...))`
Install the implementation information provided in yourself, for future reference. Reply with an indication that you have done so.

`(a summarize-implementation)`
Reply with a summary of the implementation information previously installed within you. Encapsulate that information in an instance description of the form:

```
(an installation
    (with expression ...)
    (with environment ...)
    (with pattern ...))
```

`(a-concept-for-instance-description)`
Reply with a concept appropriate for an instance description being created. The reply contains either yourself or your name, depending upon what your creation information indicates.

`(an are-you (with what ...))`
Recognizes concepts `'atomic-description`, `'description`, and `'concept`.

E.10 Instance Descriptions

Instance descriptions obey the common protocols, as well as the protocols below. Expect the set of protocols obeyed by descriptions in general to increase when inheritance and deduction mechanisms are embedded in Act2.

`(a make-descriptor (with environment ...))`
Reply with an instance description which has the same concept as you do, but which has no attributions. This is typically used for extracting type information for an actor from a description of it, which was used in its creation.

`(an are-you (with what ...))`
Recognizes the concepts `'instance-description` and `'description`.

Appendix F

Other Language Issues

F.1 Lexical Scoping

Act2 implements lexical scoping of free identifiers in abstraction definitions. These are conceptually more appropriate for programmers in general [Sussman and Steele 75, Church 41, Landin 64]. It has the property of referential transparency. That is, if a programmer defines an abstraction with free variables, those free identifiers are resolved in the definition environment. There will be no accidental name conflicts with code which instantiates the abstraction [Sussman and Steele 75].

Lexical scoping and static binding are essential for controlled sharing and authentication. They guarantee that the expressions denoting patterns in an abstraction definition are evaluated in the definition environment. The atomic descriptions used for the concepts will be those in the definition environment. Only those atomic descriptions conforming to those will match, so our authentication mechanism is preserved. If free identifiers were bound dynamically, as in Lisp, these authentication mechanisms would not work.

F.2 Aliasing

Act2 realizes the actor computational model, in which actors are independent virtual computational agents. Identifiers in Act2 serve as *names* for denoting actors, and not as information containers, such as identifiers in languages such as Fortran. A single actor can be referred to with different names in a computation. Modern object-oriented languages tend to favor this approach of using identifiers as names,

rather than as containers [Liskov, et al 81, Ingalls 78].

The notion of sharing fostered by this view of identifiers has further implications. There is no need to distinguish between parameter-passing techniques. Container-oriented languages have notions of call by reference, call by value, etc., which determine the relationship between identifiers used as formal and as actual parameters. These typically affect the meaning of assignment within the abstraction body. In Act2, identifiers are names, and there are no assignment commands, so this problem does not arise. The parameter-passing semantics are those of call-by-sharing, as in Clu [Liskov, et al 81].

F.3 No Identifier Lifetime Problems

Act2 has no lifetime, or dangling reference, problems. Actors exist as long as they are accessible, so no dangling references can occur. Act2 inherits this property from the underlying Apiary architecture, which is responsible for storage management. The Apiary performs garbage collections to reclaim storage associated with inaccessible actors. This is a major benefit for Act2 programmers, because they do not need to be concerned with allocation and deallocation of storage. Programmers using languages without garbage collection typically spend a large fraction of their time thinking about storage management in their programs. Act2 programmers are spared from this time-consuming activity.

F.4 Context Sensitivity

Act2 contains several context-sensitive commands and expressions, to increase the conciseness, readability, and programmability of the language. Some were introduced into the language in order to reduce the verbosity of the language by omitting information which is obvious to a reader, or which is available form the local context. This also is very convenient for the writer of Act2 code.

For example, a create expression appearing as an abstraction declaration, such as

```
(define (new checking-account (with balance ≡b) (with owner ≡o))
  (create ...))
```

becomes associated with a description, `(a checking-account)`, which serves as its "type".

The **new** expression is also context-sensitive. When appearing inside a become command with the sole effect of changing some acquaintances, without changing its script, it needs only to mention those needing change. For example, `(become (new checking-account (with balance 60)))`, when appearing in a context such as that above, would be equivalent to `(become (new checking-account (with balance 60) (with owner charles)))`.

The become command is also context-sensitive in another way. For it to be truly context-free, it would have to indicate not only the replacement actor, but also the actor to replace, which happens to be the one in whose script the **become** command appears. No more than one become command can be evaluated in response to a communication. Also, the become command is context-sensitive in the sense that it is not allowed to appear in composite-expression bodies.

The **reply** and **complain** commands are very context sensitive. The target for

the reply or complaint communication is left unspecified and must be identified by Act2 from context. When appearing as a command in a request-handler context, a reply command sends its communication to the customer from the request, complaints are sent to the request's complaint department.

When appearing in the body of a composite expression, a reply or complaint command designates a reply or complaint to the evaluation of the expression. Exactly one must be evaluated in the evaluation of the expression.

Sponsors, the resource management mechanisms, are usually handled entirely by context. Typically, the sponsor contained in the communication being processed pays for the processing of that communication. None of the constructs explicitly mention sponsors, except the **using-sponsor** construct, which works by affecting the context of the commands it contains, and the `is-communication` variation of communication handlers, which allows the programmer to have a pattern which will extract the sponsor during a pattern-match.

F.5 Compilation fits into Interactive Framework

Act2 treats compilation purely as an optimization technique. The ideal is that programmers should not be able to tell the difference between compiled and uncompiled code. Separate compilation is supported at the abstraction level. The protocol for compilation is, in the case of **factorial**,

```
(ask factorial (a compile)).
```

Bibliography

[Attardi, Simi 81]
Attardi, G. and Simi, M.
Semantics of Inheritance and Attributions in the Description System Omega.
In *Proceedings of IJCAI 81*. IJCAI, Vancouver, B. C., Canada, August, 1981.

[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.

[Church 41]
Church, A.
The Calculi of Lambda-Conversion.
In *Annals of Mathematics Studies Number 6.* Princeton University Press, 1941.

[Clinger 81a]
Clinger, W.D.
Foundations of Actor Semantics.
PhD thesis, Massachusetts Institute of Technology, May, 1981.
Available as MIT AI Lab TR 633

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

[Dennis 81]
Dennis, J.B.
Data Should Not Change: A Model for a Computer System.
Available as MIT Laboratory for Computer Science Computer Structures Group Memo 209.

[Guttag, Horowitz and Musser 76]
Guttag, J.V., E. Horowitz, and D.R. Musser.
Abstract Data Types and Software Validation.
Technical Report RR-76/48, USC/Information Sciences Institute, August, 1976.

[Hewitt 77]
Hewitt C.
Viewing Control Structures as Patterns of Passing Messages.
Artificial Intelligence 8:323-364, 1977.
Also available as MIT AI Memo 410

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

[Hewitt and Attardi 81]
Hewitt, C.E. and G. Attardi.
Guardians for Concurrent Systems.
Draft of MIT Artificial Intelligence Laboratory memo.

[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 and Smith 75]
Hewitt. C., and B. Smith.
Towards a Programming Apprentice.
IEEE Transactions on Software Engineering SE-1(1), march, 1975.

[Hewitt, Attardi, Lieberman 79]
Hewitt C., Attardi G., and Lieberman H.
Specifying and Proving Properties of Guardians for Distributed Systems.
In *Proceedings of the Conference on Semantics of Concurrent Computation.* INRIA, Evian, France, July, 1979.

[Hewitt, Attardi, Simi 80]
Hewitt, C., Attardi, G., and Simi, M.
Knowledge Embedding with a Description System.
In *Proceedings of the First National Annual Conference on Artificial Intelligence.* American Association for Artificial Intelligence, August, 1980.

[Hewitt, de Jong 82]
Hewitt, C., de Jong, P.
Open Systems.
In Brodie, M. L., Mylopoulos, J. L., Schmidt, J. W., editor, *Perspectives on Conceptual Modeling.* Springer-Verlag, 1982.

[Ingalls 78]
Ingalls, D.H.H.
The Smalltalk-76 Programming System: Design and Implementation.
In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages.* Association for Computing Machinery, 1978.

[Jefferson, Sowizral 82]
Jefferson, D., Sowizral, H.
Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control.
Technical Report N-1906-AF, RAND, December, 1982.

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

[Kornfeld 79]
Kornfeld, W.
Using Parallel Processing for Problem Solving.
AI Memo 561, MIT, December, 1979.

[Kornfeld 82]
Kornfeld, W.
Concepts in Parallel Problem Solving.
PhD thesis, Massachusetts Institute of Technology, 1982.

[Kornfeld, Hewitt 81]
Kornfeld, W. A. and Hewitt, C.
The Scientific Community Metaphor.
IEEE Transactions on Systems, Man, and Cybernetics SMC-11(1), January, 1981.

[Landin 64]
Landin, P.J.
The Mechanical Evaluation of Expressions.
The Computer Journal 6(4), January, 1964.

[Lieberman 81a]
Lieberman, H.
A Preview of Act-1.
A.I. Memo 625, MIT Artificial Intelligence Laboratory, 1981.

[Lieberman 81b]
Lieberman, H.
Thinking About Lots of Things At Once Without Getting Confused: Parallelism in Act-1.
A.I. Memo 626, MIT Artificial Intelligence Laboratory, 1981.

[Lieberman 82]
Lieberman, H.
Personal communication.

[Lieberman 83]
Lieberman, H.
An Object-Oriented Simulator for the Apiary.
Draft of MIT Artificial Intelligence Laboratory memo.

[Lieberman and Hewitt 83]
Lieberman, H., and C. Hewitt.
A Real Time Garbage Collector Based on the Lifetimes of Objects.
Communications of the ACM , June, 1983.
Also available as MIT AI Memo 569A

[Liskov 72]
Liskov, B.H.
A Design Methodology for Reliable Software Systems.
AFIPS Conference Proceedings 41 1:191-199, 1972.
FJCC

[Liskov, et al 81]
G. Goos and J. Hartmanis, editor.
Lecture Notes in Computer Science. Volume 114: *CLU Reference Manual.*
Springer-Verlag, New York, 1981.
Also available as MIT LCS TR 225, October 1979.

[Sandewall 80]
Sandewall, E.
Programming in the Interactive Environment: the Lisp Experience.
Computing Surveys 10(1), March, 1980.

[Sussman and Steele 75]
Sussman, G.J. and G.L. Steele.
SCHEME: An Interpreter for Extended Lambda Calculus.
Available as MIT AI Memo 349

[Theriault 82]
Theriault, D.
A Primer for the Act1 Language.
A.I. Memo 672, MIT Artificial Intelligence Laboratory, April, 1982.

[Turner 79]
Turner, D.A.
A New Implementation Technique for Applicative Languages.
Software - Practice and Experience 9:31-49, 1979.

[Waters 83]
Waters, R.C.
Lets: An Expressional Loop Notation.
Available as MIT AI Memo 680A

[Weinreb and Moon 81]
Weinreb, D., and D. Moon.
Lisp Machine Manual.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 2/15/96

Report # AI-TR-728

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 218 (226 - IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☒ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): FOLLOWS TITLE, ABST, ACK, DED, & LAST PAGE (213)

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:

IMAGE MAP: (1-218) (UNNUM'D TITLE, BLANK, ABSTRACT, BLANK, ACK, BLANK, DEDICATION, BLANK, 5-213, BLANK (UNNUM'D) (219-226) SCANCONTROL, COVER, SPINE, DOD(2), TRGT(3)

Scanning Agent Signoff:
Date Received: 2/15/96 Date Scanned: 2/21/96 Date Returned: 2/29/96

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR 728	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Issues in the Design and Implementation of Act 2		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* Daniel G. Theriault		8. CONTRACT OR GRANT NUMBER*(s)* N0014-80-C-0505
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE June 1983 13. NUMBER OF PAGES 213
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

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

18. SUPPLEMENTARY NOTES

None

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

Actor Languages	Open Systems
Description and Action	Concurrent Systems
Computer Programming Languages	Distributed Systems
Message Passing Semantics	Parallelism

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

Act2 is a highly concurrent programming language designed to exploit the processing power available from parallel computer architectures. The language supports advanced concepts in software engineering, providing high-level constructs suitable for implementing artificially intelligent applications. Act2 is based on the Actor model of computation, consisting of virtual computational agents which communicate by message-passing. Act2 serves as a framework in which to integrate an actor language, a description and reasoning system, and a problem-solving and resource management system. This document describes

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT NUMBER	GOVT ACCESSION NO.	RECIPIENT'S CATALOG NUMBER
AI-TR 718		

TITLE (and Subtitle)	TYPE OF REPORT & PERIOD COVERED
Issues in the Design and Implementation of Act 2	Technical Report
	PERFORMING ORG. REPORT NUMBER

AUTHOR(s)	CONTRACT OR GRANT NUMBER(s)
Daniel G. Theriault	N00014-80-C-0505

PERFORMING ORGANIZATION NAME AND ADDRESS	PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139	

CONTROLLING OFFICE NAME AND ADDRESS	REPORT DATE
Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209	June 1983
	NUMBER OF PAGES 213

MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	SECURITY CLASS. (of this report)
Office of Naval Research Information Systems Arlington, Virginia 22217	UNCLASSIFIED
	DECLASSIFICATION/DOWNGRADING SCHEDULE

DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

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

SUPPLEMENTARY NOTES

None

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

Actor Languages	Open Systems
Description and Action	Concurrent Systems
Computer Programming Languages	Distributed Systems
Message Passing Semantics	Parallelism

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

Act 2 is a highly concurrent programming language designed to exploit the processing power available from parallel computer architectures. The language supports advanced concepts in software engineering, providing high-level constructs suitable for implementing artificially-intelligent agent applications. Act 2 is based on the Actor model of computation, combining of virtual computational agents which communicate by message-passing. A description in Act 2 of a problem-solving and resource-management system will integrate an actor language, a description and [illegible]

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**