An Overview of Actor Languages

Gul Agha
The Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

June 28, 1986

Abstract

Actors are computational agents which carry out their actions in response to incoming communications. As in other object-oriented languages, actors encapsulate procedural and declarative information into a single entity. Primitive actor languages are based on a pure message-passing semantics. Higher-level actor languages incorporate the use of ***inheritance*** for conceptual organization and ***delegation*** for structuring the sharing of code between different actors. This paper provides an overview of the actor model and discusses some advantages of actor languages in exploiting large-scale concurrent architectures. The kernel of an actor language is presented and some of the higher-level control constructs are discussed.

1 Motivation

In this section, we discuss some of the salient characteristics of actor-based languages and compare them to other programming languages. Specifically, we discuss issues of encapsulation, inheritance, delegation and concurrency. In the following sections, we will elaborate on the concept of actors and show some examples which illustrate the flavor of actor systems.

[*] The ideas discussed in this paper have been stimulated by discussions with Carl Hewitt, Peter de Jong, Carl Manning, Tom Reinhardt, and Toon-King Wong. The work described in this paper was supported by in part by a grant from the System Development Foundation. Support for the laboratory's artificial intelligence research is also provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N0014-80-C-0505.

1 MOTIVATION

1.1 Encapsulation

Objects provide an elegant mechanism for modular decomposition of real-world systems. By encapsulating information into autonomous components, actor programs, unlike logic-based languages, are capable of incorporating different and often contradictory viewpoints without compromising the integrity of the system as a whole [Hew85]. Since the behavior of an actor can be affected only by sending it a communication, actor languages maintain privacy and security with respect to the internal structure of the individual actors.

The semantics of object-based languages is an area that has not received enough attention; an object is usually defined only in intuitive terms. Early development of the concept of an object can be traced to Simula where an object was thought of as a self-contained program having its own data and actions [BDMN73]. In Smalltalk an object contained both a private memory and a set of operations which could be validly applied to the contents of that memory [GR83]. Actors share the same containment property characterizing objects in Simula and Smalltalk.

1.2 Inheritance

A specific scheme for inheritance is not germane to actor languages. By contrast, Simula and Smalltalk have the notion of a ***class object*** to which objects belong. Furthermore, in Smalltalk the classes themselves are considered objects belonging to meta-classes. The notion of classes and meta-classes provides a mechanism for sharing information between different objects via ***inheritance***. However, inheritance is not the only scheme for information sharing, and in any case, the requirement that each object permanently belong to a class imposes constraints on the mutability of the behavior of an object. The notion of a class is not integral to the actor model; consequently, an actor may transform its behavior without necessarily being constrained by restrictions imposed due to membership in a given class.

Inheritance provides a means of conceptually organizing information about the world. One particularly attractive scheme for organizing knowledge is in terms of ***generalizations*** and ***specializations*** along a lattice. The description system OMEGA [AS81], which has been implemented in terms of actors, organizes descriptions in OMEGA along a binary lattice; the disjunction of two descriptions represents their generalization and the conjunction, their specialization. The top of the lattice is a universal description (`something`) and the bottom of the lattice is a contradiction that cannot describe any object (`nothing`). For example, the conjunction of `French` and `Automobile` is the description `French Automobile` which will have properties satisfied by those descriptions. In case of contradictions, an actor implementation may rely on specific contradiction handlers. The results of the conflict resolution can be subsequently installed in the lattice.

One advantage of programming with actors is that actors permit the description hierarchies to be dynamically reconfigurable; thus the system is capable of conceptually re-organizing itself as it interacts with its environment. In OMEGA-like languages, the hierarchies are of arbitrary depth; thus actors permit more complex taxanomies than those afforded by a single-layered class structure which differentiates between classes and objects contained in them.

1.3 Delegation

Another mechanism for information sharing is *delegation* [HRAA85]. Using delegation, sub-computations can be passed on by an actor to another actor which continues the processing. Delegation does not help organize knowledge in terms of a taxonomy, but provides a mechanism for code-sharing whereby the control is actually passed to an independent actor. By contrast, in inheritance mechanisms information may be requested from a more general class to which an actor belongs; however, the control remains localized. Specifically, an actor delegating a computation simply sends a request to the *delegate* and can itself proceed to accept further communications. Delegation promotes modularity of code by avoiding the need to duplicate the same code in the body of different actors.

1.4 Concurrency

Development of the actor model has been guided by the goal of exploiting large-scale concurrency [AH85]. Actors are computational agents which may function in parallel. The idea behind actor languages is to provide the syntactic constructs that can free the programmer from having to worry about the details of a concurrent execution. In particular, it is important to remove unnecessary data dependencies created by the assignment command (the so-called von Neumann bottleneck). One proposal for avoiding the proliferation of data dependencies is that of functional programming [Bac78]. By excluding assignments to a store, functional programming allows for the possibility of concurrent evaluation of expressions in a program. Unfortunately, functional programming is unable to address the problem of ***history-sensitive*** behaviors. Dataflow architectures (for example [AA82]) have addressed the issue of history-sensitivity by streams with recursive feedback on nodes with functional behavior.

Actor languages also avoid the assignment command but allow actors to specify a *replacement*. Replacement in actors has two implications. On the one hand, actor languages are able to capture history-sensitive information, and on the other, they allow for concurrent evaluation of expressions that do not involve a necessary data dependency [Agh86]. The advantage of using actors to model *open systems* is that actors, unlike nodes in a dataflow graph, are dynamically reconfigurable; as a consequence, actors are suitable for representing the continuing evolution common to real world systems.

2 Actor Systems

Actor systems represent a community of actors. Some actors serve as ***receptionists*** to the system and may receive communications from outside the system. Because the existence of other actors may be communicated to actors outside the system, the set of receptionists changes as the system evolves. Actors communicate by sending each other messages. The mail system provides asynchrony and buffering of communications between actors. Communication in actors is point-to-point: the sender must specify a specific target to which the communication is to be sent.

2.1 What is an Actor?

Actors were first defined simply in terms of the relationship between events caused by an actor [Hew77]. Early work in actor semantics developed the laws of parallel processing; such laws are general axioms that must be satisfied by all distributed systems [HB77,Cli81]. The current formulation of the actor model identifies the fundamental constructs necessary for an actor system. A higher-order language has been defined in terms of these primitives. A representation has been developed which hides information internal to actor systems and allows one to compose them using message-passing [Agh86]. Concurrency control issues such as deadlock and divergence have also been addressed in the new framework [Agh86,Agh85]. On the implementation side, a class of actor languages has been implemented. These languages treat continuations as first-class objects and use message-passing to implement control structures in concurrent actor languages [Hew77,The82,The83].

An actor has a mail address and a behavior. The mail address of an actor may be freely communicated—a feature which results both in the ability to reconfigure the system, and in the ability to extend a system (since mail addresses from the outside may be communicated). In response to processing a communication targeted to an actor, the behavior of an actor consists of three kinds of actions (see Figure 1):

1. An actor may ***send communications*** to specific actors it knows the mail address of. In particular, an actor may send communications to itself.

2. An actor may ***create new actors***. Initially, the mail address of such actors may be known only to the creator and possibly to the actor itself. However, the mail address can be subsequently communicated.

3. An actor <u>must</u> ***specify a replacement*** which will accept the next communication. The replacement may process the next communication even as other actions occurring as a result of processing the previous communication are still being executed.

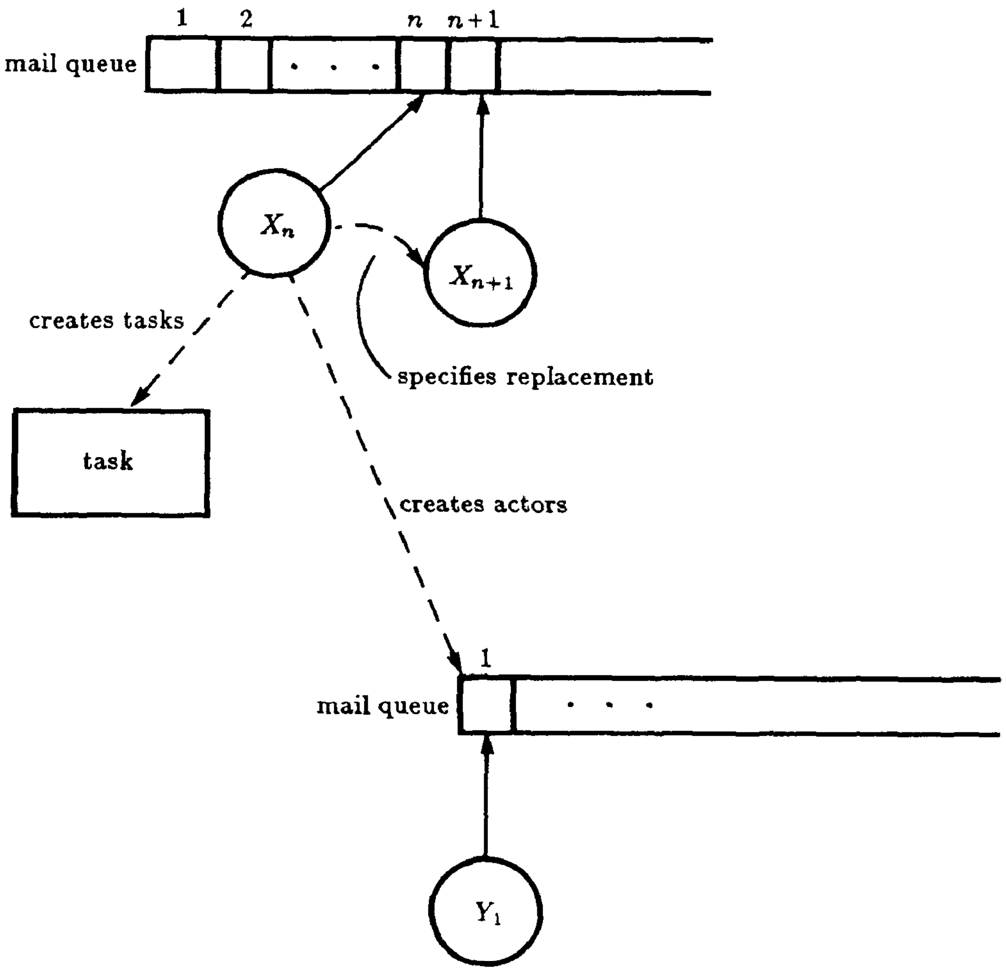

Figure 1: *An abstract representation of the structure of an actor and its possible actions. A task is a communication together with its target actor. As the figure illustrates, the replacement process may be pipelined.*

Concurrency is spawned in an actor system by sending a number of messages in response to a single message. Furthermore, computation is distributed by the creation of customers which represent the continuations and function concurrently with the actor that created them. For example, a recursive factorial is implemented in terms of an actor which, when it receives a request to evaluate the factorial of n, creates a customer to wait for a message giving the factorial of $n - 1$. The factorial actor then sends itself a request to evaluate the factorial of $n - 1$ together with the address of the customer. Thus, a stack is unfolded as a chain of customers to reduce sequential bottlenecks in the computation.

Replacement differs from a change of local state—the replacement does not update the variables of the old state. Thus the replacement process can be pipelined so that an actor can concurrently process more than one communication. In particular, if the behavior of an actor is ***history-insensitive***, it is possible for an actor to specify its replacement before carrying out any computation dependent upon the communication received.

2.2 Message-Passing Semantics

There are two components to the environment which provides a closure in which the body of an actor's code is executed. First, mail addresses of actors that an actor knows about are bound to identifiers in the code of the latter; such actors are called *acquaintances*. Second, bindings of identifiers in the body of an actor are provided by the incoming communication.

There are two kinds of actors in the closure—*local* and *external*. Actors which are typically local include numbers, amounts, literals, etc. For example, a checking-account actor may have an acquaintance which is the balance in the account; the balance may be local: in this case, only the amount of the balance and not its mail address can be meaningfully communicated to another actor. Communicating the mail address of the balance is not meaningful because the balance is never updated: when the actor specifies a replacement, it specifies a new behavior which has a new actor representing the balance in the account.

The checking-account actor may know of an associated savings-account. In this case, a checking-account may:

- send a communication to the savings account; or
- send the mail address of the savings-account to another actor.

In any case, the checking-account has no access to the "current behavior" of the savings account. In particular, the behavior of the savings-account may change before it accepts a communication from an actor. In other words, because communication in actors is asynchronous, the behavior of actors at invocation time may differ from the behavior at execution time. Message-passing in actors, viewed as a parameter-passing mechanism, differs both from call-by-value and call-by-reference. In case of local actors, it behaves like call-by-value would and in case of external actors, it behaves like call-by-reference.

3 An Actor Language

In this section, we give the syntax of an actor language. We first define a kernel language which is sufficient to incorporate actor constructs and then define some higher-level constructs which make programming simpler in some cases.

3.1 A Minimal Language

The *Act* programming language incorporates the minimal constructs necessary to program arbitrary actor systems. An actor program consists of a sequence of *behavior definitions* followed by a *command*. Each behavior definition provides a template with which actor behaviors can be defined. A behavior is given by specifying an identifier bound to a behavior definition and values for the acquaintances in the given behavior definition.

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

There are five kinds of commands. The *send command* is used to send communications. The result of the send command is to send the value of the second expression to the target specified by the first expression. The *let command* binds expressions to identifiers in the body of commands nested within their scope. The *conditional command* provides a mechanism for branching, and the *become command* specifies the replacement behavior. Note that actors are created by *new expressions* which is the keyword new followed by a behavior. Finally, the *concurrent composition* of commands is also a command; specifically, commands specified in the code are, by default, executed concurrently.

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

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

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

```
⟨send command⟩ ::= (send ⟨expression⟩ ⟨expression⟩)

⟨become command⟩ ::= (become ⟨expression⟩)
```

We give the simple example of the code for a factorial actor. This example illustrates, among other things, the use of message-passing to implement control structures. Note that a recursive factorial distributes the work to *customers* so that different requests to a factorial can be evaluated concurently.

```
(define (Factorial( ))
  (Is-Communication (a eval (with customer ≡c)
                            (with number ≡n)) do
     (become Factorial)
     (if (NOT (= n 0))
       (then (send m 1))
       (else (let (x (new FactCust (with customer c)
                                   (with number n)))
                (send Factorial (a eval (with customer x)
                                   (with number n-1))))))))

(define (FactCust (with customer ≡m)
                  (with number ≡n))
    (Is-Communication (a number k) do
       (send m n*k)))
```

A more parallel algorithm for evaluating factorial is in terms of a range-product which continually subdivides the problem and concurrently evaluates the two sub-products creating customers to carry out the multiplications. The creation of customers is an important tool in avoiding bottlenecks; unlike multiple activations of a process, this solution also works for actors whose behavior is *serialized* (i.e. is history-sensitive).

3.2 Some Linguistic Constructs

We describe some higher-level constructs which make it easier to program in actors. Generally, the compiler of an actor language provides for the automatic generation of the appropriate customers. For example, the recursive factorial function shown above can be written in an actor language as:

```
(define (call Factorial (with number ≡n))
    (if (= n 0)
       (then 1)
       (else (* n (call Factorial (with number n-1))))))
```

Note that all sub-expressions in actor languages are, by default, evaluated concurrently. Thus, a number of customers are created to serve as *joins* for the results of a computation. Sequential composition of commands is also implemented by creating customers to carry out the rest of the computation. Note that sequential composition is meaningful only because *call expressions* impose synchronization requirements, otherwise sequentiality is meaningless given that the arrival order of communications is arbitrary in the first place [Agh86].

A higher-level language can also provide for eager evaluation using *futures*. Instead of waiting for an expression to be evaluated, the expression is treated as an actor and sent a communication to evaluate itself; the mail address of the expression actor is then used in place of the actor. Whenever the value of the expression is required, the expression actor is sent a `who-are-you` request. Communications sent to the expression are, however, buffered until the expression has evaluated itself. Futures provide a mechanism by which computation can proceed concurrently with the evaluation of the expressions which may at some point be needed in the computation. An advantage of an evaluation strategy using futures is that local convergence of a computation, if it exists, can be guaranteed. Specifically, if the value of the expression is never needed, the fact that it never completes evaluation will not affect the rest of the computation.

4 Conclusions

Actors are useful for general purpose programming and are particularly useful for Artificial Intelligence applications. The actor model provides fine-grained concurrency. Higher-level actor languages provide tools for conceptual organization and structuring; at the same time, actor languages free the programmer from concerns about the details of a program's concurrent execution.

References

[AA82] T. Agerwala and Arvind. Data flow systems. *Computer*, 15(2), Feb 1982.

[Agh85] G. Agha. A message-passing paradigm for object management. *IEEE Database Engineering Bulletin*, Dec 1985.

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[AH85] G. Agha and C. Hewitt. Concurrent programming using actors: exploiting large-scale parallelism. In *Proceedings of Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS, Springer-Verlag, 1985.

REFERENCES

[AS81] G. Attardi and M. Simi. Semantics of inheritance and attributions in the description system omega. In *Proceedings of IJCAI 81*, IJCAI, Vancouver, B. C., Canada, August 1981.

[Bac78] J. Backus. 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.

[BDMN73] G. M. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Van Nostrand Reinhold, New York, 1973.

[Cli81] W. D. Clinger. *Foundations of Actor Semantics*. AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Publishing Company, 1983.

[HB77] C. Hewitt and H. Baker. Laws for communicating parallel processes. In *1977 IFIP Congress Proceedings*, pages 987–992, IFIP, August 1977.

[Hew77] C.E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8-3:323–364, June 1977.

[Hew85] C. Hewitt. The challenge of open systems. *Byte*, 10(4):223–242, April 1985.

[HRAA85] C. Hewitt, T. Reinhardt, G. Agha, and G. Attardi. Linguistic support of serializers for shared resources. In *Seminar on Concurrency*, pages 330–359, Springer-Verlag, 1985.

[The82] D. Theriault. *A Primer for the Act-1 Language*. A.I. Memo 672, MIT Artificial Intelligence Laboratory, April 1982.

[The83] D. Theriault. *Issues in the Design and Implementation of Act2*. Technical Report 728, MIT Artificial Intelligence Laboratory, June 1983.