<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author</td>
<td>L. G. Agha et al.</td>
</tr>
<tr>
<td>Institution</td>
<td>Cambridge Artificial Intelligence</td>
</tr>
<tr>
<td>Title</td>
<td>Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism</td>
</tr>
<tr>
<td>Date</td>
<td>07 Oct 85</td>
</tr>
<tr>
<td>Number</td>
<td>AI-M-865</td>
</tr>
<tr>
<td>Classification</td>
<td>UNCLASSIFIED</td>
</tr>
<tr>
<td>Classification Code</td>
<td>00014-90-C-0505</td>
</tr>
<tr>
<td>Access Level</td>
<td>F/G 9/2</td>
</tr>
<tr>
<td>Pages</td>
<td>1/1</td>
</tr>
</tbody>
</table>
Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism

Gul Agha
Carl Hewitt

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

October 7, 1985

Abstract

We argue that the ability to model shared objects with changing local states, dynamic reconfigurability, and inherent parallelism are desirable properties of any model of concurrency. The actor model addresses these issues in a uniform framework. This paper briefly describes the concurrent programming language Act$ and the principles that have guided its development. Act$ advances the state of the art in programming languages by combining the advantages of object-oriented programming with those of functional programming. We also discuss considerations relevant to large-scale parallelism in the context of open systems, and define an abstract model which establishes the equivalence of systems defined by actor programs.

The report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the the System Development Foundation and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N0014-80-C-0505. The authors acknowledge helpful comments from Fanya Montalvo, Carl Manning and Tom Reinhardt.
Concurrent Programming Using Actors: Exploiting Large Scale Parallelism

Gul Agha
Carl Hewitt

Artificial Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139

Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

Office of Naval Research
Arlington, VA 22217

Distribution is unlimited.

Concurrency
Distributed Computing
Programming Languages
Object-Oriented Programming

Functional Programming
Parallel Processing
Open Systems

We argue that the ability to model shared objects with changing local states, dynamic reconfigurability, and inherent parallelism are desirable properties of any model of concurrency. The actor model addresses these issues in a uniform framework. This paper briefly describes the concurrent programming language Act3 and the principles that have guided its development. Act3 advances the state of the art in programming languages by combining the advantages of object-oriented programming with those of functional programming. We also discuss considerations relevant to the large scale parallelism in the
1 Background

The theory of concurrent programming languages has been an exciting area of research in the last decade. Although no consensus has emerged on a single model of concurrency, many advances have been made in the development of various contending models. There have also been some consistent paradigm shifts in the approach to concurrency; an interesting discussion of such paradigm shifts may be found in [Pratt 83].

The actor model of computation has developed contemporaneously in the last decade along with other models based on Petri Nets, the $\lambda$-calculus, and communicating sequential processes. There has been a great deal of useful cross fertilization between the various schools of thought in addressing the very difficult issues of concurrent systems. Over the years Hoare, Kahn, MacQueen, Milner, Petri, Plotkin, and Pratt, have provided fruitful interaction on the development of the actor model.

Landin [65] first showed how Algol 60 programs could be represented in applicative-order $\lambda$-calculus. Kahn and MacQueen [77] developed this area further by expanding on the construct of streams which captured functional systems. Brock and Ackerman [77] extended the Kahn-MacQueen model with the addition of inter-stream ordering information in order to make it more suitable for concurrent computation. Pratt [82] generalized the functional model by developing a theory of processes in terms of sets of partially ordered multisets (pomsets) of events. Each pomset in Pratt's Process Model represents a trace of events. Pratt's model satisfies several properties desirable in any model of concurrent computation. For example, the model does not assume the existence of global states: a trace is only a partial order of events. Thus the model is compatible with the laws of parallel processing formulated in [Hewitt and Baker 77] and shown to be consistent in [Clinger 81].

On the practical side, McCarthy [59] first made functional programming available by developing LISP. The standard dialect of LISP now incorporates lexical scoping and closures which makes the semantics simpler and programming modular [Steele, et al 84]. Act3 generalizes the lexical scoping and upward closures of LISP in the context of parallel systems.

Hoare [78] proposed a language for concurrency, called CSP, based on sequential processes. CSP, like Act3, enhances modularity by not permitting any shared variable between processes; instead, communication is the primitive by which processes may affect each other. At a more theoretical level, Milner [80] has proposed the Calculus of Concurrent Systems (CCS).
2 THE ACTOR MODEL

One of the nice properties of CCS is its elegant algebraic operations. In both CSP and CCS, communication is synchronous and resembles a handshake. In contradistinction, the actor model postulates the existence of a mail system which buffers communication.

The plan of this paper is as follows: the first section outlines the actor model. The second section describes the ActS language. The final section discusses the general principles of open systems and their relation to the actor model.

2 The Actor Model

In this section we motivate the primitives of the actor model. We will outline the basic issues and describe a set of minimal constructs necessary for an actor language.

2.1 Foundational Issues

A number of difficult open problems and foundational issues in the design of programming languages for concurrent systems merit attention. We consider the following three significant:

1. Shared Resources. The programming model must deal with the problem of shared resources which may change their internal state. A simple example of such an object in a concurrent environment is a shared bank account. Purely functional systems, unlike object-based systems, are incapable of implementing such objects [Hewitt, et al 84].

2. Dynamic Reconfigurability. The programming model must deal with the creation of new objects in the evolution of the system. In particular, to accommodate the creation of new objects, there must be a mechanism for communicating the existence of such new objects (or processes) to already existing ones. Thus when a bank creates a new account, it should be able to inform its book-keeping process of the existence of such an account. Since the interconnection topology of processes is static in systems such as CSP and dataflow [Brock 83], this requirement is necessarily violated these systems.

3. Inherent Parallelism. The programming model should exhibit inherent parallelism in the sense that the amount of available concur-
rency should be clear from the structure of programs. It should not be necessary to do extensive reasoning to uncover implicit parallelism that is hidden by inappropriate language constructs. In particular, the assignment command is a bottleneck inherited from the von Neumann architecture. Assignment commands tie the statements in the body of a code in such a way that only through flow analysis is it possible to determine which statements can be executed concurrently. Functional Programming has the advantage of being inherently parallel because it allows the possibility of concurrent execution of all subexpressions in a program [Backus 78].

The object-based and functional, $\lambda$-calculus-based languages represent two of the most important schools of thought in programming language theory today. As the above discussion suggests, both have certain advantages. Act3 attempts to integrate both in a manner that preserves some of their attractive features.

2.2 Basic Constructs

The actor abstraction has been developed to exploit message-passing as a basis for concurrent computation [Hewitt 77; Hewitt and Baker 77]. The actor construct has been formalized by providing a mathematical definition for the behavior of an actor system [Agha 85]. Essentially, an actor is a computational agent which carries out its actions in response to processing a communication. The actions it may perform are:

- Send communications to itself or to other actors.
- Create more actors.
- Specify the replacement behavior.

In order to send a communication, the sender must specify a mail address, called the target. The mail system buffers the communication until it can be delivered to the target. However, the order in which the communications are delivered is nondeterministic. The buffering of communications has the consequence that actor languages support recursion. In languages relying on synchronous communication, any recursive procedure immediately leads to deadlock [Hewitt, et al 1984] [Agha 1985].
2 THE ACTOR MODEL

All actors have their own (unique) mail addresses which may be communicated to other actors just as any other value. Thus mail addresses provide a simple mechanism for dynamically reconfiguring a system of actors. The only way to affect the behavior of an actor is to send it a communication. When an actor accepts a communication, it carries out the actions specified by its behavior; one of these actions is to specify a replacement actor which will then accept the next communication received at the mail address.

Two important observations need to be made about replacement. First, replacement implements local state change while preserving referential transparency of the identifiers used in a program. An identifier for an object always denotes that object although the behavior associated with the object may be subject to change. In particular, the code for an actor does not contain spurious variables to which different values are assigned (see [Stoy 77] for a thorough discussion of referential transparency). Second, since the computation of a replacement actor is an action which may be carried out concurrently with other actions performed by an actor, the replacement process is intrinsically concurrent. The replacement actor cannot affect the behavior of the replaced actor.

The net result of these properties of replacement actors is that computation in actor systems can be speeded-up by pipelining the actions to be performed. As soon as the replacement actor has been computed, the next communication can be processed even as other actions implied by the current communication are still being carried out. In actor-based architectures, the only constraints on the speed of execution stem from the logical dependencies in the computation and the limitations imposed by the hardware resources. In von Neumann architectures, the data dependencies caused by assignments to a global store restrict the degree of pipelining (in the form of instruction pre-fetching) that can be realized [Hwang and Briggs 84].

All actors in a system carry out their actions concurrently. In particular, this has the implication that message-passing can be used to spawn concurrency: An actor, in response to a communication, may send several communications to other actors. The creation of new actors also increases the amount of parallelism feasible in a system. Specifically, continuations can be incorporated as first-class objects. The dynamic creation of customers in actor systems (discussed later) provides a parallel analogue to such continuations.
2 THE ACTOR MODEL

2.3 Transitions on Configurations

To describe an actor system, we need to specify several components. In particular, we must specify the behaviors associated with the mail addresses internal to the system. This is done by specifying a local states function which basically gives us the behavior of each mail address (i.e., its response to the next communication it receives). We must also specify the unprocessed communications together with their targets. The communication and target pairs are referred to as tasks. A configuration is an instantaneous snapshot of an actor system from some viewpoint. Each configuration has the following parts:

- A local states function which basically gives us the behavior of a mail address. The actors whose behaviors are specified by the local states function are elements of the population.

- A set of unprocessed tasks for communications which have been sent but not yet accepted.

- A subset of the population, called receptionist actors, which may receive communications from actors outside the configuration. The set of receptionists can not be mechanically determined from the local states function of a configuration: it must be specified using knowledge about the larger environment.

- A set of external actors whose behavior is not specified by the local states function, but to whom communications may be sent.

A fundamental transition relation on configurations can be defined by applying the behavior function of the target of some unprocessed task to the communication contained in that task (see the definition below). Given the nondeterminism in the arrival order of communications, this transition relation represents the different possible paths a computation may take. The processing of communications may, of course, overlap in time. We represent only the acceptance of a communication as an event. Different transition paths may be observed by different viewpoints, provided that these paths are consistent with each other (i.e. do not violate constraints such as causality).

Definition 1 Possible Transition. Let \( c_1 \) and \( c_2 \) be two configurations. \( c_1 \) is said to have a possible transition to \( c_2 \) by processing a task \( r \), symbolically, \( c_1 \rightarrow r c_2 \) if \( r \in \text{tasks}(c_1) \), and furthermore, if \( \alpha \) is the target of the
3 THE ACT3 LANGUAGE

of the task then the tasks in \( c_3 \) are

\[
\text{tasks}(c_3) = (\text{tasks}(c_1) - \{r\}) \cup T
\]

where \( T \) is the set of tasks created by \( \alpha \) in response to \( r \), and the actors in \( c_3 \) are

\[
\text{actors}(c_3) = (\text{actors}(c_1) - \{\alpha\}) \cup A \cup \{\alpha'\}
\]

where \( A \) are the actors created by \( \alpha \) in response to \( r \) and \( \alpha' \) is the replacement specified by \( \alpha \). Note that \( \alpha \) and \( \alpha' \) have the same mail address.

In the actor model, the delivery of all communications is guaranteed. This form of fairness can be expressed by defining a second transition relation which is based on processing finite sets of tasks until a particular task is processed, instead of simply processing a single task [Agha 84]. A denotational semantics for actors can be defined in terms of the transition relations; this semantics maps actor programs into the initial configuration they define [Agha 85].

3 The Act3 Language

\textit{Act3} is an actor-based programming language which has been implemented on the \textit{Apiary architecture}. The Apiary is a parallel architecture based on a network of Lisp machines and supports features such as dynamic load balancing, real-time garbage collection, and the mail system abstraction [Hewitt 80]. \textit{Act3} is a descendant of \textit{Act2} [Theriault 83] and is written in a LISP-based interface language called \textit{Scripter}.

A program in \textit{Act3} is a collection of behavior definitions and commands to create actors and send communications to them. A behavior definition consists of an identifier (by which the actor may be known), a list of the names of acquaintances, and a script (which defines the behavior of the actor in response to the communication it accepts). When an actor is created its acquaintances must be specified. For example, a bank-account actor may have an acquaintance representing its current balance.

When a communication is accepted by an actor, an environment is defined in which the script of the actor is to be executed. The commands in the script of an actor can be executed in parallel. Thus \textit{Act3} differs fundamentally from programming languages based on communicating sequential processes since the commands in the body of such processes must be executed sequentially.
We will first provide the syntax for a kernel language, *Act*, and use it to explain the basic concepts of message-passing. We then discuss some extensions to *Act* which are provided in *Act*$. Finally, we illustrate these extensions by means of examples.

3.1 The Kernel Language Act

The language *Act* is a sufficient kernel for the *Act*$ language: all constructs in the *Act*$ language can be translated into *Act* [Agha 85]. Since there are so few constructs in *Act*, it will be easier to understand the primitives involved by studying *Act*. The acquaintance list in *Act* is specified by using identifiers which match a pattern. The pattern provides for freedom from *positional* correspondence when new actors are created. Patterns are used in pattern matching to bind identifiers, and authenticate and extract information from data structures. The simplest pattern is a *bind pattern* which literally binds the value of an identifier to the value of an expression in the current environment. We will not concern ourselves with other patterns here.

When an actor accepts a communication it is *pattern-matched* with the *communication handlers* in the actor's code and dispatched to the handler of the pattern it satisfies. The bindings for the communication list are extracted by the pattern matching as well. The syntax of behavior definitions in *Act* programs is given below.

\[
\begin{align*}
\text{(act program)} & ::= \\
& \quad \text{(behavior definition)* ((command)*)} \\
\text{(behavior definition)} & ::= \\
& \quad \text{(define (id ((with identifier (pattern)))* (communication handler)*)} \\
\text{(communication handler)} & ::= \\
& \quad \text{(Is-Communication (pattern) do (command)*)}
\end{align*}
\]

The syntax of commands to create actors and send communications is the same in actor definitions as their syntax at the program level. There are four kinds of commands; we describe these in turn. *send commands* are used to send communications. The syntax of the *send command* is the keyword *send* followed by two expressions: The two expressions are evaluated; the
first expression must evaluate to a mail address while the second may have an arbitrary value. The result of the `send` command is to send the value of the second expression to the target specified by the first expression. `let commands` bind expressions to identifiers in the body of commands nested within their scope. In particular, `let commands` are used to bind the mail addresses of newly created actors. `new expressions` create new actors and return their mail address. A `new expression` is given by the keyword `new` followed by an identifier representing a behavior definition, and a list of acquaintances.

The `conditional command` provides a mechanism for branching, and the `become command` specifies the replacement actor. The expression in the `become command` may be a `new expression` in which case the actor becomes a forwarding actor to the actor created by the `new expression`; in this case the two actors are equivalent in a very strong sense. The expression can also be the mail address of an existing actor, in which case all communications sent to the replaced actor are forwarded to the existing actor.

A Recursive Factorial. We first provide a simple factorial example to illustrate the use of message-passing in actors to implement control structures. The code makes the low level detail in the execution of an actor language explicit. We will subsequently provide some higher-level constructs which will make the expression of programs easier. The factorial actor creates `customers`, called FactCust, whose behavior is also given below. Note that the behavior of a factorial is `unserialized`, i.e., it is not history sensitive.
(define (Factorial( ))
  (Is-Communication (a doit (with customer =m) (with number =n)) do
    (become Factorial)
    (if (NOT (= n 0))
      (then (send m 1))
      (else (let (x = (new FactCust (with customer =m) (with number n)))
        (send Factorial (a do (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)))

The acceptance of a communication containing an integer by Factorial causes n to be bound to the integer and concurrently for factorial to become "itself" so that it can immediately process another integer without any interaction with the processing of the integer it has just received. When the factorial actor processes a communication with a non-zero integer, n, it will:

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

The customer created by the factorial actor is also an independent actor. The work done to compute a factorial is conceptually distributed by the creation of the customer. In particular, this implies that computation can be speeded-up if several factorials are to be evaluated concurrently. In the case of the factorial, the same result can be obtained by multiple activations of a given function. However, the solution using multiple activations does not work if the behavior of an actor is serialized.

3.2 Functional Constructs

In this section we will develop some notation for representing expressions at a higher-level. Act3 provides many such constructs which make Act3 far more expressive than Act, although the two languages have the same
expressive power. To allow functional programming without forcing the programmer to explicitly create the customers, Act3 provides call expressions which automatically create a customer and include its mail address in the communication sent; the value of the expression is returned (in a message) to the customer created at the time of the call. The code below specifies a factorial actor in expressional terms. By comparing the code to that in the previous section, one can see how it is executed in an actor-based environment.

\[
\text{(define (call Factorial (with number \equiv n))}
\]
\[
\begin{cases}
\text{(if (= n 0)} & \text{1} \\
\text{(else (* n (call Factorial (with number n-1))))}
\end{cases}
\]

Parallel control structures can also be specified quite easily. For example, a parallel algorithm for evaluating the factorial function of \( n \) is by recursively subdividing the problem of computing the range product from 1 to \( n \). We define an actor, RangeProduct, for recursively computing the range product in the above manner. The code for RangeProduct is given below. Note that the One-of construct provides a generalized conditional command: it dispatches on the value of the expressions (cf. the guarded command [Dijkstra 76]).

\[
\text{(define (call RangeProduct (with low \equiv lo))}
\]
\[
\text{(with High \equiv hi))}
\]
\[
\text{(One-of)}
\]
\[
\begin{cases}
\text{(if (= lo hi) lo)} \\
\text{(if (> lo hi) 1)} \\
\text{(if (< lo hi)}
\]
\[
\begin{cases}
\text{(let ((mid (/ (+ lo hi) 2)))}
\text{(* (call RangeProduct (with low lo))}
\text{(with high mid))
\text{(call RangeProduct (with low (+ mid 1))}
\text{(with high hi))})}
\end{cases}
\]

The pipelining of the replacement actors implies that two calls to the RangeProduct actor are in fact equivalent to creating two actors which function concurrently. This equivalence follows from the unserialized nature of the behavior: In case the behavior is unserialized, the behavior of the replacement is known immediately and thus its computation is immediate; in particular, it can be computed even before a communication is received.
3 THE ACT3 LANGUAGE

Act3 provides a number of other expressional constructs, such as delayed expressions and allows one to require lazy or eager evaluation strategies for expressions. Such evaluation strategies have been used in extensions of pure functional programming to model history-sensitive behavior [Henderson 80]. However, because these systems lack a mail address abstraction, the interconnection network topology of processes is entirely static.

3.3 Modelling Local-State Change

A problem with functional programming is the difficulty of dealing with shared objects which have changing local states. Some constructs, such as delayed expressions have been defined to model changing local states. However, the problem with these techniques is that they create expressional forms totally local to the caller and thus can not be used to represent shared objects. Actors permit a graceful implementation of shared objects with a changing local state. The example below shows the implementation of a bank account in Act3. A bank account is a canonical example of a shared object with a changing local state.

We use the keyword Is-Request to indicate a request communication is expected. A request communication comes with the mail address of the customer to which the reply is to be sent. The customer is used as the target of the reply. A request also specifies a mail address to which a complaint can be sent, should the request be unsuccessful. From a software point of view, providing independent targets for the complaint messages is extremely useful because it allows the error-handling to be separated from successfully completed transactions.

(define (Account (with Balance ≡b))
  (Is-Request (a Balance) do (reply b))
  (Is-Request (a Deposit (with Amount ≡a)) do
    (become (Account (with Balance (+ b a))))
    (reply (a Deposit-Receipt (with Amount a))))
  (Is-Request (a Withdrawal (with Amount ≡a)) do
    (if (> a b)
      (then do (complain (an Overdraft)))
      (else do
        (become (Account (with Balance (- b a))))
        (reply (a Withdrawal-Receipt (with Amount a)))))
)
Note that the `become` command is pipelined so that a replacement is available as soon as the `become` command is executed. The commands for other actions are executed concurrently and do not affect the replacement actor which will be free to accept further communications.

### 3.4 Transactional Constructs

Analyzing the behavior of a typical program in terms of all the transitions it makes is not very feasible. In particular, the development of debugging tools and resource management techniques requires us to preserve the abstractions in the source programs. Because actors may represent shared objects, it is often critical that transitions relevant to independent computations be kept separate. For example, if the factorial actor we defined is asked to evaluate the factorial of \(-1\), it will create an "infinite loop." Two observations should be made about such potentially infinite computations. First, any other requests to the factorial will not be affected because the guarantee of delivery means that communications related to those requests will be interleaved with the "infinite loop" generated by the \(-1\) message. Second, in order to keep the performance of the system from degrading, we must assess costs for each "computation" independently; we can then cut-off those computations that we do not want to support indefinitely.

To formalize the notion of a "computation," we define the concept of transactions. Transactions are delineated using two specific kinds of communications, namely, requests and replies. A request, \(r_1\), may trigger another request, say \(r_2\); if the reply to \(r_2\) also precedes the reply to \(r_1\), then the second transaction is said to be nested within the first. Proper nesting of transactions allows simpler resource management schemes since resources can be allocated dynamically for the sub-transaction directly from the triggering transaction.

Transactions also permit the development of debugging tools that allow one to examine a computation at different levels of granularity [Manning 84]. Various constructs in Acts permit proper nesting of transactions; for example, requests may be buffered while simultaneously preserving the current state of a server using a construct called enqueue. The request is subsequently processed, when the server is free to do so, using a dequeue operation. Enqueue and dequeue are useful for programming servers such as those controlling a hard copy device; they guarantee continuous availability [Hewitt et al 1984].

Independent transactions may affect each other; requests may be sent
4 OPEN SYSTEMS

4.1 Characteristics of Open Systems

We list three important considerations which are relevant to any architecture supporting large-scale parallelism in open systems [Hewitt 85]. These considerations have model theoretic implications for an algebra used to characterize the behavior of actors:

- **Continuous Availability.** A system may receive communications from the external environment at any point in time. There is no closed-world hypothesis.

- **Modularity.** The inner workings of one subsystem are not available to the any other system; there is an arms-length relationship between subsystems. The behavior of a system must be characterized only in terms of its interaction with the outside.

- **Extensibility.** It is possible for a system to grow. In particular, it is possible to compose different systems in order to define larger systems.

Actors provide an ideal means of realizing open systems. In the section below, we outline a model which realizes the above characteristics and, at the same time, abstracts the internal events in an actor system. We thus address the problem of abstraction in the context of open system modelling.
4 OPEN SYSTEMS

4.2 A Calculus of Configurations

We have described two transition relations on configurations (see §2.3). These relations are, however, operational rather than extensional in nature. The requirements of modularity imply that an abstract characterization of the behavior of an actor system must be in terms of communications received from outside the system and those sent to the external actors. All communications sent by actors within a population, to other actors also within the population, are not observable from the outside.

In the denotational semantics of sequential programming languages, it is sufficient to represent a program by its input-output behavior, or more completely, as a map from an initial state to a final state (the so-called history relation). However, in any program involving concurrency and nondeterminism, the history relation is not a sufficient characterization. Specifically, when two systems with identical history relations are each composed with an identical system, the two resulting systems have different history relations [Brock and Ackerman 81]. The reason for this anomaly is the closed-world assumption inherent in the history relation: It ignores the possible interactions of the output with the input [Agha 85].

The fact that communications may be accepted from the outside at any point. There are three kinds of derivations from a configuration:

1. A configuration $c$ is said to have a derivation to $c'$ given an input task $r$, symbolically, $c \xrightarrow{\uparrow} c'$, if

   \[
   \begin{align*}
   \text{states}(c') &= \text{states}(c) \\
   \text{tasks}(c') &= \text{tasks}(c) \cup r \land \text{target}(r) \in \text{population}(c)
   \end{align*}
   \]

   where \text{states} represents the local states function (see §2.3), and \text{tasks} represents the tasks in a configuration. The receptionists remain the same but the external actors may now include any actors whose mail addresses have been communicated by the communication accepted.

2. A configuration $c$ is said to have a derivation to $c'$ producing an output task $r$, symbolically, $c \xrightarrow{\downarrow} c'$, if

   \[
   \begin{align*}
   \text{states}(c') &= \text{states}(c) \\
   \text{tasks}(c') &= \text{tasks}(c) - r \land \text{target}(r) \notin \text{population}(c)
   \end{align*}
   \]

   where the \text{states} and the \text{tasks} are as above, and \(\sim\) represents set theoretic difference. The external actors of $c'$ are the same as those of
c. The receptionists may now include all actors whose mail addresses have been communicated to the outside.

3. A configuration \( c \) has an internal or silent derivation to a configuration \( c' \), symbolically, \( c \xrightarrow{\sigma} c' \), if it has a possible transition to \( c' \) for some task \( r \) in \( c \).

We can now build a calculus of configurations by defining operations such as composition, relabeling (which changes the mail addresses), restriction (which removes a receptionist), etc. We give the axioms of compositionality to illustrate the calculus of configurations.

**Definition 2 Composition.** Let \( c_1 \parallel c_2 \) represent the (concurrent) composition of \( c_1 \) and \( c_2 \). Then we have the following rules of derivation about the composition:

1. (a) Let \( r \) be a task whose target is in \( c_1 \), then

\[
\begin{align*}
  c_1 \xrightarrow{c_1} c_1', \quad c_2 \xrightarrow{c_2} c_2' \\
  \hline
  c_1 \parallel c_2 \xrightarrow{c_1' \parallel c_2'}
\end{align*}
\]

(b) Let \( \lambda \) be any derivation (input, output, or internal), provided that if \( \lambda \) is an input or output derivation then its sender or target, respectively, is not an actor in \( c_1 \), then

\[
\begin{align*}
  c_1 \xrightarrow{\lambda} c_1' \\
  \hline
  c_1 \parallel c_2 \xrightarrow{\lambda} c_1' \parallel c_2
\end{align*}
\]

2. The above rules hold, mutatis mutandis, for \( c_2 \parallel c_1 \).

The only behavior that can be observed in a system is represented by the “labels” on the derivations from its configurations. These represent the communications between a system and its external environment. Following Milner [80] we can define an observation equivalence relation on configurations. The definition relies on equality of all possible finite sequences of communications sent to or received from the external environment (ignoring all internal derivations). One way of formalizing observation equivalence is inductively:
Definition 3 Observation Equivalence. Let $c_1$ and $c_2$ be any two tasks, $\mu$ be either an input or an output task, $e^*$ represent any arbitrary (finite) number of internal transitions, and $\stackrel{e^*}{\rightarrow}$ represent a sequence of internal transitions followed by a $\mu$ transition, and furthermore $\approx_k$ be defined inductively as:

1. $c_1 \approx_0 c_2$
2. $c_1 \approx_{k+1} c_2$ if
   
   $\forall \mu (if \ c_1 \stackrel{e^*}{\rightarrow} c_1 \ then \ \exists c'_1 (c_2 \stackrel{\mu}{\rightarrow} c'_1) \land c_1 \approx_k c'_1)$

$Now c_1$ is said to be observationally equivalent to $c_2$, symbolically, $c_1 \approx c_2$, if $\forall k (c_1 \approx_k c_2)$.

The notion of observation equivalence is weaker than that of the history relation—it creates fewer equivalence classes and thus distinguishes between more configurations. Specifically, it allows for distinguishing between systems that behave differently in response to new tasks, after having sent some communication to an external actor.

We can characterize actor programs by the equivalence classes of initial configurations they define. Properties of actor system can be established in a framework not relying on a closed-world assumption, while at the same time providing an abstract representation of actor systems that does not rely on the internal details of a systems behavior.

5 Conclusions

Actor languages uniformly use message-passing to spawn concurrency and are inherently parallel. The mail system abstraction permits a high-level mechanism for achieving dynamic reconfigurability. The problem of shared resources with changing local state is dealt with by providing an object-oriented environment without the sequential bottle-neck caused by assignment commands. The behavior of an actor is defined in Acts by a script which can be abstractly represented as a mathematical function. It is our claim that Acts has the major advantages of object-based programming languages together with those of functional and applicative programming languages.
An actor language also provides a suitable basis for large-scale parallelism. Besides the ability to distribute the work required in the course of a computation, actor systems can be composed simply by passing messages between them. The internal workings of an actor system are not available to any other system. A suitable model to support the composition of different systems is obtained by composing the configurations they may be in.
References


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


[Landin 65] Landin, P. A Correspondence Between ALGOL 60 and Church's Lambda Notation. Communication of the ACM 8, 2 (February 1965).


