North-Holland
Microprocessing and Microprogramming 18 (1986) 307–318

MESSAGE PASSING AND GRAPH TRANSFORMATIONS : A MODEL OF ACTOR COMPUTATION

D. Janssens

Bell Telephone Mfg Co., TA4
Francis Wellesplein 1
B-2018 ANTWERP
BELGIUM

Actor systems are a model of massively parallel computation. In an actor system a computation is performed by a number of independent active elements, called actors. Communication between these actors happens by asynchronous message passing. The aim of this paper is to develop a comprehensive description of actor systems, based on techniques from the area of graph grammars. We will start with a somewhat oversimplified model and subsequently enrich it as we focus on certain basic features of actor systems. Programming an actor system will correspond to writing productions, which are then to be applied to a graphical representation of the global state of the system. We will also pay attention to various implementation issues, such as memory management, load balancing, etc.

1. ACTOR SYSTEMS : INTRODUCTION

Actor systems have been introduced as a model of parallel computation. The model is based on asynchronous message passing between a (large) number of independently active computing agents, called actors.

It is motivated by recent developments in both software and hardware : on the one hand, the model naturally supports the principles of object oriented programming and, on the other hand, development of massively parallel architectures creates the need for suitable models of computation, allowing for massive, medium or fine grain parallellism.

The basic elements of an actor system are actors and messages. Actors are active objects which are able to receive messages and to react to them in three ways (which can be combined) : (1) they can send new messages to other actors, (2) they can change their local state and (3) they can create new actors.
The action of accepting a message and reacting to it (in the way described above) is considered the "atomic action" of the system : we will not go under this level of abstraction, and the reaction will be considered (or, at least, described) as instantaneous).

The properties of the model are determined to a large extent by the assumptions one makes about the communication between actors. A first assumption is that the communication is completely asynchronous : the sender of a message does not "know" when its message will be received by the destination actor; once a message is sent, the sender "forgets" it and may start reacting to the next message it receives. Evidently, this implies that messages are buffered at their destination. We assume no upper bound on the size of the buffers. Also, delivery of all messages sent is guaranteed. Another important assumption concerning the communication between actors is that each actor can send messages only to a limited number of other actors; these are called its acquaintances.
A last assumption concerns the order in which messages arrive. In the terminology of [1], the processing of a message by an actor is called an "event". Since an actor cannot receive several messages simultaneously, the

events corresponding to a given target actor occur in a certain linear order. This order is called the arrival ordering of the given actor. We will assume that the arrival ordering is arbitrary; in other words, that we do not have any information about the delay of the messages.

Since one generally assumes that the number of actors which are simultaneously active is quite high, the model offers good prospects with respect to its implementation on a massively parallel architecture : a "program" may consist of several hundreds or thousands of actors, which can be distributed over a large number of physical processors. Since communication becomes a central issue in such an architecture, one tries to make the assumptions on the communication facilities such that they can be (relatively) easily supported : this is one of the reasons why we prefer asynchronous communication over synchronous communication, we allow arbitrary arrival orderings and we restrict actors to sending messages to a limited number of other actors only.

An important aspect of actor systems is their dynamic nature. If one considers the processing of a message by an actor as an atomic event, then a message is, in a sense, equivalent to an instruction in a procedural model. (On this level, the assumption that delivery of messages is guaranteed becomes very natural). One of the possible responses to a message is the creation of new actors. Hence, in contrast with some other models of concurrent computation, such as CSP or Petri Nets, the creation of new "processes" (actors) is itself part of the model. Moreover, if one assumes that messages may contain information about actors, then an actor may change its acquaintances in reaction to a message received.

Actor systems have been investigated by several authors, such as [1-4]. Experimental programming languages based on the actor paradigm have been proposed, such as ACT I, II and III (see [3] and [5]). Our aim is to develop a comprehensive, mathematical description of an actor system and its operation. To this aim we will represent an instantaneous state of an actor system by a graph, and apply ideas from graph grammars to model its evolution in time. (A graph grammar is a system describing the generation of graphs by applying productions, in a way analogous to the generation of strings by formal grammars. (See, e.g. [6]).

2. A GRAPH-GRAMMAR MODEL FOR ACTOR SYSTEMS

2.1 Basic definitions

If we want to model the operation of an actor system by the application of graph grammar productions to graphs representing instantaneous states or "configurations" of the system, then the first thing to do is to define this representation. The representation should describe at least the following elements :

- the actors present in the system,
- the local state of each actor,
- the acquaintaince-relation,
- the messages present.

In our first approach we will assume that messages are simply elements of a fixed, finite set of possible messages. This set will be denoted by $\mathcal{M}$. (If one wants to model the fact that a message contains information about other actors - a communication list, in the terminology of [4] - then it may be desirable to change this). Furthermore, we assume that the behaviour of each actor (i.e., its reaction

to a given message) is determined by one of a finite set of "local states". The set of all possible states (for all actors) is denoted by $\mathscr{A}$. If one wants to consider different types of actors, then one can assume that the sets of possible states for different kinds of actors are disjoint. Hence, in that case, the different types lead to a partition of $\mathscr{A}$.

Throughout the rest of the paper we assume that $\mathscr{A}$ and $\mathscr{M}$ are fixed finite nonempty sets.

Formally, a configuration may be defined as a system (AC, E, st, msg), where AC is a finite set (of actors), E is a subset of AC x AC, st is a function from AC into $\mathscr{A}$ and msg is a function from AC into the set of multisets over $\mathscr{M}$. Hence, the representation we will use is a node and edge labeled directed graph; the nodes represent actors, an edge from A_1 to A_2 represents the fact that A_2 is an acquaintance of A_1, the function st represents the local states of the actors and the function msg represents the messages sent but not yet received.

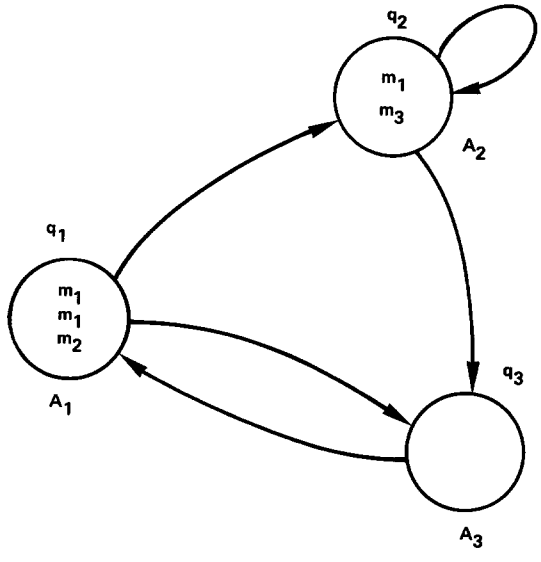


Fig. 1

In fig. 1 an example of such a configuration is depicted : the system contains three actors, A_1, A_2, A_3, with local states q_1, q_2 and q_3. A_1 has two acquaintances : A_2 and A_3. A_1 has three messages waiting to be received : two occurrences of m_1 and one occurrence of m_2.

Remark

The fact that messages are represented by elements of a multiset associated to their destination actor does not imply that we assume their immediate delivery in a physical implementation : the presence of such a message only means that it is on its way to its destination and that it will eventually be received.

As a step towards modeling the dynamic behaviour of the system, we will describe the behaviour of systems in which no new actors are created and the acquaintance relation remains unchanged.

In such systems, an "elementary step" consists of two stages :

- First one has to choose which actors will receive a message, and which message they receive. Since we do not want to have global synchronization in the system, this choice will happen nondeterministically.
- Second, the selected actors react to the messages received by changing their local states and sending new messages. They do so on the basis of two elements : their local state and the message received.

The selection of messages to be received will be described by a selection function and the way actors react to messages will be specified by the behaviour function.

In the first approach, one might define the behaviour function to be a function from

$AC \times \mathcal{M}$ into $\mathcal{A} \times \mathcal{P}(AC \times \mathcal{M})$, where the first component of the image would specify the new state and the second component would be a set of pairs (A, m), indicating which messages (m) are sent to which actors (A). However, we want the behaviour to be dependent on the local state of an actor rather than on the actor itself : two actors having the same state should react identically when receiving the same message. Therefore the behaviour function should be a function from $\mathcal{A} \times \mathcal{M}$ rather than from $AC \times \mathcal{M}$. This does not solve the problem completely, however : two actors have, in general, different acquaintances, even if they have the same local state. One should be able to distinguish between the acquaintances of an actor without having to list them explicitely. One way of doing this is to index the acquaintances of an actor or, equivalently, the outgoing edges of a node in a configuration. We will denote the maximum number of acquaintances of an actor in the system by the letter n. We will also use the following notions and notation.

- If n is a positive integer, then N denotes the set {1, 2, ... n}.
- Let U and W be finite nonempty sets. A directed labeled graph over X and Y is a system G = (V, E, lab) where V is a finite nonempty set, $E \subseteq V \times W \times V$ and lab is a function from V into U.
- If G = (V, E, lab) is a directed labeled graph and e = (x, w, y) $\in$ E, then s(e) = x, ind (e) = w and t(e) = y.
- If G = (V, E, lab) is a directed labeled graph and v $\in$ V, then
  $v^\bullet = \{e \in E \mid s(e) = v\}$ and
  $^\bullet v = \{e \in E \mid t(e) = v\}$.

Throughout the paper we will use the following extended notion of a configuration.

Definition 1. An n-configuration is a directed labeled graph C = (AC, E, lab) over $\mathcal{A} \times BAG(\mathcal{M})$ and N (where $BAG(\mathcal{M})$ is the set of multisets over $\mathcal{M}$) such that, for each x $\in$ AC, the restriction of ind to $x^\bullet$ is injective (i.e., two outgoing edges of a node must have different indices).

Conventions and notation

(1) Throughout the paper we will assume a fixed maximal number of acquaintances, n, and we will use the term "configuration" instead of "n-configuration".
(2) In specifying a configuration, we will always use two functions, st and msg, to specify the first and the second component of lab, respectively. Hence st yields the local states of the actors and msg yields the bags of messages.
(3) If A and B are actors in a configuration and there exists an edge e from A to B with ind(e) = i, then B will be called the i-acquaintance of A.
(4) Throughout the rest of the paper, nil will denote a symbol not in $\mathcal{M}$, and $\bar{\mathcal{M}} = \mathcal{M} \cup \{nil\}$.

We are now ready to define the notions of a selection function and a behaviour function.

Definition 2. Let C = (AC, E, st, msg) be a configuration.

(1) A selection function on C is a partial function sel from AC into $\mathcal{M}$ such that, for each v $\in$ AC, sel(v) $\in$ msg(v).
(2) A behaviour function on C is a function beh from $\mathcal{A} \times \mathcal{M}$ into $\mathcal{A} \times (\bar{\mathcal{M}})^n$.

Notation :
We will often use the notation newst (a, m) to denote $proj_1$ (beh (a, m)). For j $\in$ N, by out_j (a, m) we denote the j-th component of $proj_2$ (beh (a, m)).

Definition 3. Let $C_1 = (AC_1, E_1, \underline{st}_1, \underline{msg}_1)$ and $C_2 = (AC_2, E_2, \underline{st}_2, \underline{msg}_2)$ be configurations and let sel be a selection function on C_1.
Then C_1 derives C_2 through sel, denoted $C_1 \overset{sel}{\Rightarrow} C_2$, if the following holds :

(1) $AC_1 = AC_2$ and $E_1 = E_2$,
(2) for each $A \in \underline{dom}\ (\underline{sel})$, $\underline{st}_2(A) = \underline{newst}\ (\underline{st}_1(A), \underline{sel}(A))$ and for each $A \in AC_1 - \underline{dom}\ (\underline{sel})$, $\underline{st}_1(A) = \underline{st}_2(A)$.
(3) for each $A \in \underline{dom}\ (\underline{sel})$, $\underline{msg}_2\ (A) = \underline{msg}_1\ (A) - \{\underline{sel}(A)\} + MES(A)$ and, for each $A \in AC_1 - \underline{dom}\ (sel)$, $\underline{msg}_2\ (A) = \underline{msg}_1\ (A) + MES(A)$, where MES(A) is the multiset over $\mathcal{M}$ such that, for each $m \in \mathcal{M}$, the multiplicity of m in MES(A) equals the cardinality of $\{e \in A \mid s(e) \in \underline{dom}\ (\underline{sel})$ and $\underline{out}_{\underline{ind}_1}(e)\ (\underline{st}_1\ (s(e)), \underline{sel}\ (s(e)) = m\}$

Remark
(1) Let C_1 and C_2 be configurations. We write $C_1 \Rightarrow C_2$ if there exists a selection function sel such that $C_1 \overset{sel}{\Rightarrow} C_2$.
(2) By $\overset{*}{\Rightarrow}$ we denote the transitive and reflexive closure of $\Rightarrow$.

2.2 Computations in actor systems

Having defined the "elementary step" of an actor system, we want to focus now on the way an actor system can be used to solve a certain task. Essentially, programming an actor system corresponds to specifying the behaviour function (or, in the extended version considered hereafter, the productions). A computation in the system is a sequence of derivation steps, starting from a given initial configuration. The actors present, with their local states and acquaintances, will be viewed as a data structure that is transformed by applying productions which are, on their turn, triggered by messages. Computations will in general not be deterministic, due to the fact that we do not specify which selection function will be chosen in each step.

In order to be able to define what is the result of a computation, one has to specify when a computation is considered to be finished. As messages correspond, in the actor model, to "instructions" requiring a reaction, it is straightforward to consider a computation as finished when all messages have been consumed and no new messages can be sent anymore. Hence, in our model, a configuration will be a final configuration if, for each actor A, we have $\underline{msg}(A) = \emptyset$.

As far as the implementation on a parallel architecture is concerned, this definition of a final configuration is probably not the easiest choice : it requires knowledge about the presence (or absence) of messages anywhere in the system; i.e., global knowledge. It is to be expected that the support of this kind of functionality will require special hardware provisions.

Another straightforward way of finishing a computation would consist in generating a special "stop" message and sending it to a "manager" actor that would recognize it and take the appropriate actions. However, one would still face the problem that the other actors have to warned of the fact that the computation has to terminate. Moreover, the decision of terminating a computation would have to be taken locally by one actor, which will be difficult because each actor has only a limited amount of information about the system.

It is easily seen that actors that never change their local state (i.e., actors that are in a local state q such that $\underline{newst}(q,m) = q$ for each m) correspond to "unserialized" actors, in the terminology of [5]. Detection

of unserialized actors can be helpful in implementing a suitable load-balancing algorithm (which is needed to maintain a reasonable distribution of the actors over the available physical processors) : the workload of an unserialized actor can be distributed over several occurrences of the actor without affecting the overall functionality of the system.
The interpretation of an actor computation can be described by the standard diagram of fig. 2,

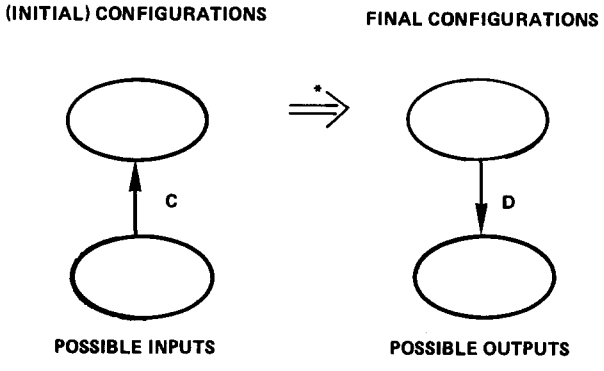


Fig. 2

where $\mathscr{C}$ is the coding of possible inputs into configurations and $\mathscr{D}$ is the decoding of final configurations into outputs.

Since computations are nondeterministic, there will in general be several final configurations $F_1, F_2, \ldots F_m$ such that, for a given configuration C, we have $C \overset{*}{\Rightarrow} F_i$ for a each $i \leq m$. If we introduce actor creation, the set of final configurations reachable from a given configuration may even become infinite. Hence, to obtain determinism on the level of input-output, one can try to choose $\mathscr{D}$ to be a constant function on the set of final configurations reachable from the initial configuration. There are several reasonable ways to achieve this : e.g., $\mathscr{D}$ can be defined such that $\mathscr{D}(F)$ depends only on actors of a certain type (or in a certain set of local states) in F.

2.3 Example

Although the model as introduced above is "static" (no creation of new actors), it allows the description of meaningful algorithms. We now describe a bounded depth, breath-first search. The search is performed on a datastructure (a tree) formed by actors and their acquaintance relation. Their local states represent a "key" and a chunk of "data". One searches for an actor for which a certain attribute (the search attribute) has a given value, which will be input to the root, and one wants to get the data of that actor. This information will be sent to a particular actor, the "customer". The finiteness of the sets $\mathscr{A}$ and $\mathscr{M}$ forces us to introduce separate local states for each possible key and each possible piece of data, but is it easily seen how this limitation can be avoided in an actual implementation.

One can describe the system as follows. Let d denote the maximal depth of the desired search. Furthermore, assume that the search attribute has k possible values and let INFO be the (finite) set of possible data elements. Then the set $\mathscr{A}$ of local states contains the following elements (and only those).

(1) For each $x \in \{1,2,\ldots,k\}$ and each info $\in$ INFO, a state $q_{x,\underline{info}}$ (corresponding to a node in the tree with search attribute x and data info).
(2) For each $j \in \{1,2,\ldots,d\}$ and each info $\in$ INFO, a state $p_{x,\underline{info}}$ (representing the fact that the customer has received a "solution" info that has been found on the j-th level of the tree).
(3) A state p_0 (the initial state of the customer).

The set $\mathcal{M}$ of possible messages contains the following elements (and only those).

(1) For each $j \in \{1,2,\ldots,d\}$ and each $x \in \{1,2,\ldots,k\}$, a message $\underline{search}_{j,x}$ (representing the fact that one searches the j-the level of the tree for actors with search attribute x).

(2) For each $j \in \{1,2,\ldots,d\}$ and each $\underline{info} \in$ INFO, a message $\underline{found}_{j,\underline{info}}$ (representing the fact that a node with the right value for its search attribute and data element $\underline{info}$ has been found at the j-th level of the tree).

We assume that the initial configuration is of the form depicted in fig. 3.

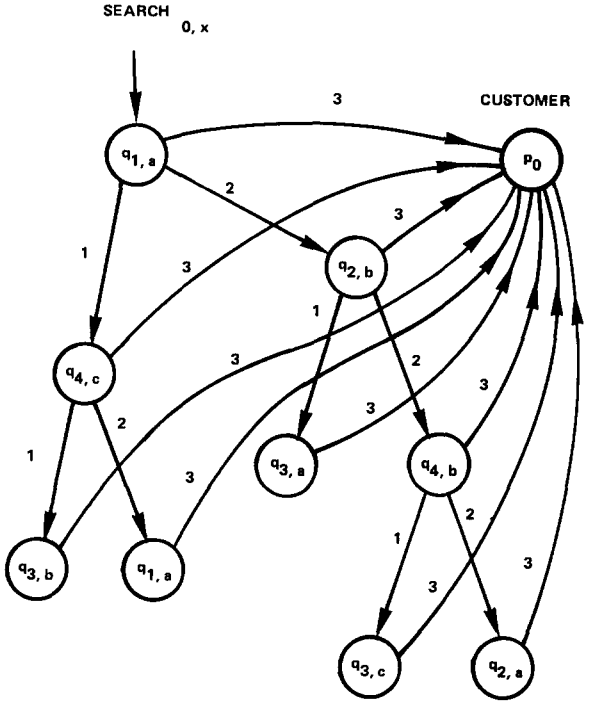


Fig. 3

On the one hand we have a number of actors, organized as a binary tree, representing the search space, and on the other hand we have a customer (which is the 3-acquaintance of every node in the tree), an actor that will collect and investigate the answers it gets from the nodes of the tree. Keys are represented by integers and "data" by small letters.

In specifying the behaviour function (which corresponds to "programming" the system) one has to ensure that the messages are used as follows :

(1) $\underline{seach}_{j,x}$ messages are propagated from the root, increasing j every time one goes one level deeper in the tree. When an actor receives such a message, it checks its search attribute and, if its value equals x, it sends a $\underline{found}_{j,\underline{info}}$ message to the customer.

(2) $\underline{found}_{j,\underline{info}}$ messages are received by the customer. It looks for the $\underline{info}$ correspondig to the minimal value of j.

Formally, the behaviour function is defined as follows :

$$\underline{beh}(q_{x,w}, \underline{search}_{j,y}) = \begin{cases} (q_{x,w}, \{3, \underline{found}_{j,w}\}), & \text{if } x=y \text{ and} \\ (q_{x,w}, \{(1, \underline{search}_{j+1,y}), (2, \underline{search}_{j+1,y})\}) & \text{if } x\neq y \text{ and } j<d \\ (q_{x,w}, \emptyset) & \text{if } x\neq y \text{ and } j=d. \end{cases}$$

$$\underline{beh}(p_0, \underline{found}_{j,u}) = (p_{j,u}, \emptyset)$$

$$\underline{beh}(p_i, \underline{found}_{j,u}) = \begin{cases} (p_{j,u}, \emptyset) & \text{if } i \leq j \\ (p_{j,u}, \emptyset) & \text{if } i>j \end{cases}$$

We disregard the specification of $\underline{beh}(q_{x,w}, \underline{found}_{j,u})$, $\underline{beh}(p_0, \underline{found}_{j,u})$ and $\underline{beh}(p_{i,w}, \underline{search}_{j,y})$, because "found" messages will only be sent to the customer and "search" messages will only be sent to actors in the tree.

Remark

Observe that the above search algorithm can easily be modified to deal with other structures than trees. Also, the optimality condition for a candidate solution (which is, in the example, minimality of the level in the tree) can be modified.

3. EXTENSION OF THE BASIC MODEL

3.1 Actor Creation

The dynamic creation of new actors is an essential part of the actor model. In term of the behaviour function, this means that, for a local state a and a message m, beh(a,m) must specify not only a new local state and a set of messages to be sent, but also the newly created actors. Moreover these new actors must get local states, acquaintances and messages. Hence the behaviour function has to specify a complete "daugher configuration" that will replace the creating actor. The creating actor will be identified with one of the actors of the daughter configuration. In the sequel we will specify the behaviour by giving a set of "productions". An elementary step in the system will consist in the (parallel) application of a number of these productions, resulting in a new configuration. The left hand side of a production is a pair from $\mathscr{A} \times \mathscr{M}$ and the right hand side will specify the messages to be sent, the "daughter configuration" and an "embedding function" yielding the acquaintances of the new actors. In order not to introduce an extra kind of nondeterminism, we assume that each pair of $\mathscr{A} \times \mathscr{M}$ occurs as the left hand side of at most one production. Formally, we have the following. Let nil be an element not in $\mathscr{M}$.

Definition 4. A production is a system $p = (a, m, \underline{out}, D, \underline{init}, \underline{emb})$, where

(1) $a \in \mathscr{A}$,
(2) $m \in \mathscr{M}$,
(3) $\underline{out} \in \{\mathscr{M} \cup \{\underline{nil}\}\}^n$;
$\underline{out} = (\underline{out}_1, \underline{out}_2, \ldots, \underline{out}_n)$,
(4) D is a configuration; let
$D = (AC_D, E_D, \underline{st}_D, \underline{msg}_D, \underline{ind}_D)$
(5) $\underline{init} \in AC_D$,
(6) emb is a partial function from $AC_D \times N$ into N such that, for each $a \in AC_D$ and $i \in N$, $(a,i) \in \underline{dom}\ (\underline{emb})$ implies that a has no i-acquaintance in D.

In the above definition, a and m specify the left-hand side of the production, out specifies the messages sent to the acquaintances of the creating actor, D is the daughter configuration, init is the actor of D which is identified with the creating actor (hence $\underline{st}_D\ (\underline{init})$ is its new local state) and emb specifies the acquaintances of the actors of D in the resulting configuration. The function emb will be used as follows : let $x \in AC_D$ and let $i \in N$. Then the i-acquaintance of x is determined as follows :

- if x has an i-acquaintance in D, y say, then y is also the i-acquaintance in the resulting configuration.
- if x does not have an i-acquaintance in D and $(x,i) \in \underline{dom}(\underline{emb})$, and $\underline{emb}(x,i)=j$, then the i-acquaintance of x in the resulting configuration is the j-acquaintance of the creating actor (in the original configuration).
- if x does not have an i-acquaintance in D and $(x,i) \notin \underline{dom}(\underline{emb})$, then x has no i-acquaintance in the resulting configuration.

Observe that the identification of the creating actor with the init actor of the daughter configuration determines what will happen to the messages of the creating actor which have not yet been received : these messages become messages of $\underline{msg}(\underline{init})$ in the resulting configuration. The choice which

messages will be accepted in an elementary step is expressed by a selection function, as in the static model.

Example

We give now an example of a derivation step in the extended, dynamic model. Let C_1 be the configuration of fig. 4, where n=3, A, B, C, D are actors,

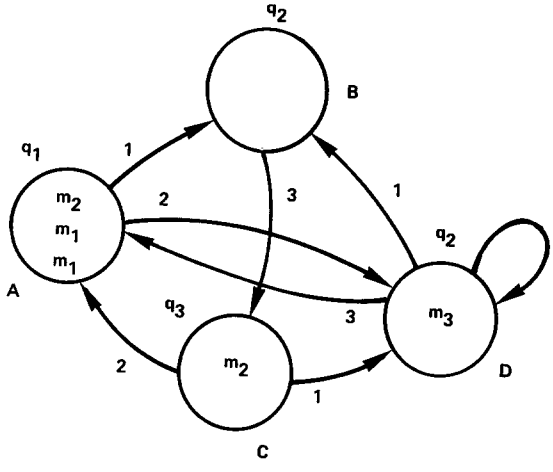
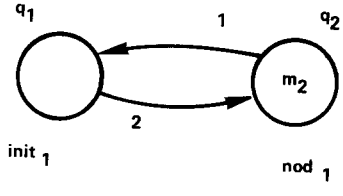


Fig. 4

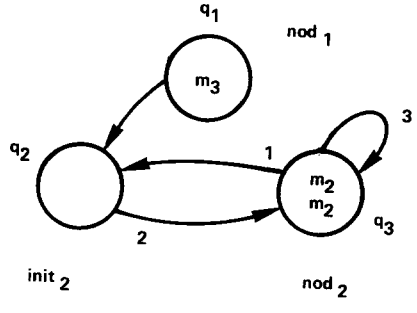
q_1, q_2, q_3 are local states and m_1, m_2, m_3 are messages.

Let $p_1=(q_1,m_2, (m_1,m_3,m_3), D_1, \underline{init}_1, \underline{emb}_1)$ and $p_2=(q_2,m_3, (\underline{nil},m_2,m_3), D_2, \underline{init}_2, \underline{emb}_2)$, where D_1, D_2, $\underline{init}_1$, $\underline{init}_2$, $\underline{emb}_1$ and $\underline{emb}_2$ are as illustrated by fig. 5.

D_1 =



D_2 =



$\underline{emb}_1$	1	2	3
$\underline{init}_1$	3	-	2
$\underline{nod}$	-	1	1

$\underline{emb}_2$	1	2	3
$\underline{init}_2$	3	-	1
$\underline{nod}_1$	-	-	2
$\underline{nod}_2$	-	3	-

Fig. 5.

If the selection function $\underline{sel}$ is such that $\underline{dom}(\underline{sel}) = \{A, D\}$, $\underline{sel}(A) = m_2$ and $\underline{sel}(D) = m_3$, then $C_1 \overset{sel}{\Rightarrow} C_2$ where C_2 is the configuration of fig. 6 (edges established according to the embedding functions are drawn in dashed lines).

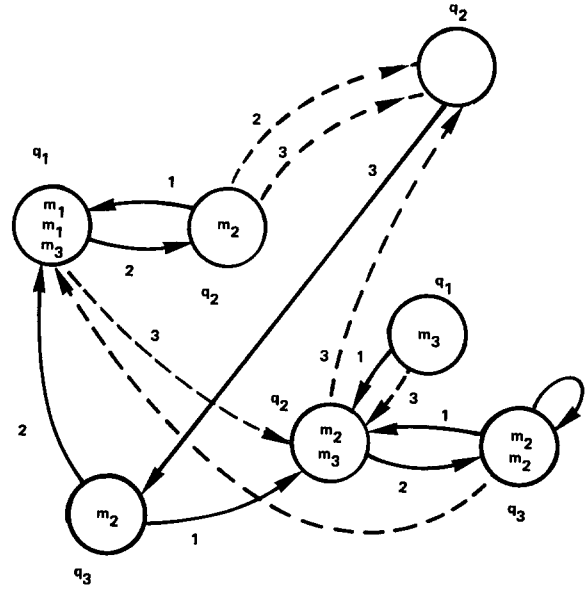


Fig. 6

3.2 Actor deletion and garbage collection.

Clearly, from a practical point of view one may expect that the creation of new actors is, in some way, compensated by the removal of "old" or "useless" actors : otherwise, the size of the system would rapidly become unmanageable. However, we do not want to give the "user" (the writer of the production) explicit control over the deletion of actors, e.g. by introducing the deletion of

an actor as a possible reaction to a message, on the same level as the creation of action or the sending of messages. The reason for this is the assumption than an actor has only information about a limited part of the system, and it does not "know" which actors will send it messages. Hence it would be strange to assume that the decision whether an actor has become useless can be taken locally.
If one has a global (a part of) view of the system, however, one can detect the actors which may be removed : those are the actors that cannot receive anymore messages in the future. It is easily seen that this is the case if the following conditions are satisfied for an actor A.

(1) msg(A) = ∅ and
(2) A does not occur among the acquaintances of any other actor.

Observe that the above conditions are sufficient but not necessary for the actor A to be useless : even if A is an acquaintance of other actors, one does not know whether they will send A any messages in the future.
The detection and removal of useless actors will be the task of a garbage collection activity, running in "background".
Taking into account the fact that the removal of one actor can make condition (2) true for another actor, we can introduce the notion of useless actors in our formal model as follows.

Definition 5. Let C=(AC,E,st,msg) be an actor configuration. An actor A ε AC is useless if

(1) msg(A) = ∅ and
(2) there exists no actor B ε AC such that msg(B) ≠ ∅ and there exists a directed path from B to A in C.

The part of the configuration consisting of useless actors will be considered as irrelevant for the computation. Clearly, in defining the decoding function $\mathcal{D}$ (defining the interpretation of a final configuration as an "output") one has to take care that $\mathcal{D}$ is not dependent on useless actors.
Also observe that there is a certain similarity between the problems of garbage collection and of detecting the termination of a computation in the sense that one needs global information about the flow of messages in the system. In order to implement algorithms to perform these tasks one will probably need extra assumptions on the communication, e.g. a bounded delay for the messages, allowing for a detection using time-outs.

3.3 Changing the acquaintance relation

In the actor systems as described so far, we considered messages to be simply elements of a finite set, $\mathcal{M}$. An essential feature of actor systems, however, is the possibility to include information about actors in the messages. (In the case of an actor system implemented on a network of physical processors, this information would correspond to the network addresses of actors). In [4], this is called the communication list of a message. An actor receiving a message including a communication list has access to all actors of both the acquaintance list (of the actor) and the communication list (of the message) : it can send messages to them, use them to change its acquaintance list or include them in the communication lists of the messages sent. Because of the analogy between the roles of both lists, we will represent the communication list in a way similar to acquaintance list : the elements of the communication list will be represented by (indexed) edges pointing to the nodes representing these elements. Of course, this

implies that messages are represented by nodes. The destination of a message will be indicated by a T-labeled edge. We will assume that the acquaintance and communication lists have the same length, n.
The configuration of fig. 4 will be represented by the graph of fig. 7, except for the dashed edges.

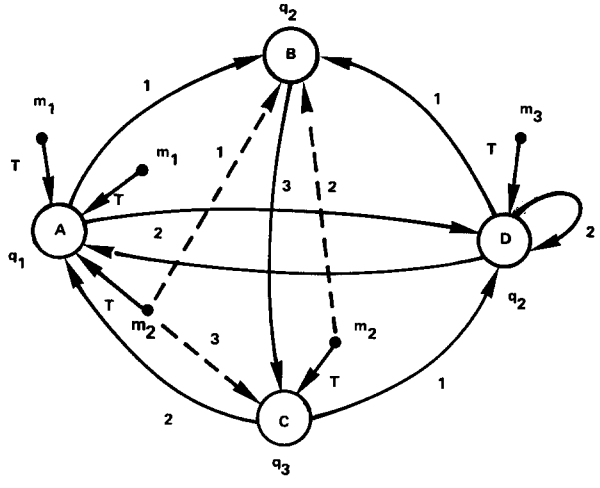


Fig. 7

The dashed edges represent the communication lists; e.g., the communication list of the m_2 message sent to A has B as its first element and C as its third element (its second element being undefined).

A derivation step will now consist of a number of parallel rewritings of two-node subgraphs, each consisting of a node representing the receiving actor, a node representing the received message and a T-labeled edge from the message to the actor. In graph grammar theory such a graph is often called a "handle", and hence our system becomes a handle rewriting system. As messages are now represented as nodes, the newly sent messages are nodes of the daughter configurations (i.e. the configurations in the right-hand sides of productions). Their destination and communication list are specified by the embedding function, which will now be a function from Nod x (N U {T}) into {1, 2, ..., 2n}, where Nod denotes the set of nodes of the daughter configuration. The use of the embedding is analogous to that of the embedding function of Section 3.2, except that beh(x,i) is now used as an index in a "context list", L, of length 2n : the concatenation of the acquaintance list of the receiving actors and the communication list of the received message. The selection function has to satisfy the following conditions :

(1) The subgraphs (handles) selected for rewriting may not overlap (an actor cannot receive more than one message at a time).
(2) The nodes in the selected handles must have "enough" elements in their acquaintance and communication list to ensure that each message has a well-defined destination : if v is a node of the daughter configuration D and the destination of v is not one of the actors of D, then the beh(v,T)-th element of L must be defined.

Conditions restricting the way in which productions may be applied, such as (1) and (2) above, are know in graph grammar theory as application conditions. As a final remark, observe that the characterization of a useless actor becomes extremely simple : an actor A is useless if there exists no message-node (a node representing a message) m such that there exists a directed path from m to A.

4. CONCLUDING REMARKS

We have presented a framework for the representation of actor systems by labeled graphs. The evolution of the system is modeled by transformations of the graph. We believe that it is worth investigating to

which extent such a model can be made into an effective actor language. Since the role of data structures is played by structures of actors, it is also important to study the graph generating power of our system. Another topic for further investigation is the assumption that the arrival ordering is arbitrary : it may be desirable to include extra assumptions on the message traffic in the model (bounded delay, order preservation of messages between two actors, etc.).

REFERENCES

[1] W.D. Clinger, Foundations of Actor Semantics, Ph.D. Thesis, Massachusetts Institute of Technology (1981). Available as technical report 633, MIT AI lab.

[2] C. Hewitt and B. Smith, Towards a Programming Apprentice, IEEE Transactions on Software Engineering (1975).

[3] D. Theriault, Issues in the Design and Implementation of ACT 2, technical report 728, MIT AI lab (1983).

[4] G.A. Agha, Actors : a Model of Concurrent Computation in Distributed Systems, technical report 844, MIT AI lab (1985).

[5] H. Lieberman, A Preview of Act 1, technical report 625, MIT AI lab (1981).

[6] H. Ehrig, M. Nagl and G. Rozenberg, Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science 153, (Springer Verlag, 1983).