Using Passive Object Garbage Collection Algorithms for Garbage Collection of Active Objects *

Abhay Vardhan
Department of Computer Science,
University of Illinois
Urbana, Illinois
vardhan@uiuc.edu

Gul Agha
Department of Computer Science,
University of Illinois
Urbana, Illinois
agha@cs.uiuc.edu

ABSTRACT

With the increasing use of active object systems, agents and concurrent object oriented languages like Java, the problem of garbage collection (GC) of unused resources has become more complex. Since active objects are autonomous computational agents, unlike passive object systems the criterion for identifying garbage in active objects cannot be based solely on reachability from a root set. This has led to development of specialized algorithms for GC of active objects. We reduce the problem of GC of active objects to that of passive objects by providing a *transformation* of the active object reference graph to a passive object reference graph so that if a garbage collector for a passive object system is applied to the transformed graph, precisely those objects are collected which correspond to garbage objects in the original active object reference graph. The transformation technique enables us to reuse the algorithms already developed for passive objects systems. We provide a proof of correctness of the transformation and discuss its cost. An advantage of the transformation is that it can prove valuable for mixed systems of active and passive objects by providing a common approach to GC.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

General Terms

Algorithms

*This research was made possible in part by support from the US Department of Defense Advanced Research Projects Agency (DARPA contract number F30602-00-2-0586) and from the Air Force Office of Scientific Research MURI contract (AFSOR contract number F49620-97-1-0382).

ISMM'02, June 20-21, 2002, Berlin, Germany.

Keywords

Garbage collection, actors, active objects, program transformation, Java, agents

1. INTRODUCTION

In distributed systems resources are often scattered across autonomous hosts making it extremely difficult to manually recognize when a particular resource can be safely reclaimed. The increasing use of active objects systems, agents and object-oriented languages like Java which provide threads, has further added to the complexity of the problem. In such systems, automating garbage collection is essential.

A substantial amount of research has been done in the area of garbage collection for functional, procedural and object-oriented languages in both sequential and distributed systems. However, it has been commonly believed that algorithms for garbage collection in these systems are not applicable directly for active object or *actor* systems. In traditional object-oriented systems, garbage collection involves a directed graph called the *reference graph* with objects as nodes and object references as edges. A subset of objects, called *root objects*, are always considered non-garbage. The problem of garbage collection is reduced to identifying and removing objects that are not reachable from any object in the root set.

In a system of actors, this is not an appropriate criterion for garbage. To see this, consider an actor which is not reachable from any root actor but is processing a message and has a reference to a root actor. According to the notion of garbage in passive object systems, such an actor should be considered garbage. Observe that although the actor is not be reachable from any root actor, the actor itself may have a reference to the root actor. Thus, this actor may send a message containing its address to the root actor it knows, thereby making itself connected from that root. Clearly, this actor cannot be considered garbage.

This paper focuses on the problem of garbage collection of active objects in a distributed environment. Actors[2] are autonomous computational agents which encapsulate data as well as some primitive processive power to manipulate the data. They offer an intuitive and powerful abstraction for concurrent and distributed systems as they reflect the real-world nature of interacting and autonomous entities.

Actors and its variants are a popular paradigm for programming in a distributed environment. Some examples of such systems include Actor Foundry[1], Act++[16], ABCL[39], Erlang[4] and Actalk[9]. There has also been a convergence

between languages developed to support other models of distributed computing, such as nomadic PICT[33] and *Join calculus*[13], and the essential constructs in the Actors model. Finally, various agent languages essentially support actor semantics.

In this paper, we describe a transformation of the actor-reference graph which captures all the information necessary for actor GC, and makes it possible to apply a garbage collection algorithm for passive objects to the transformed graph in order to collect garbage actors. The transformation represents each actor in the original graph by a pair of nodes in the transformed graph. References between nodes in the transformed graph are derived using rules which depend not only on the actors to which know a particular actor, but also on which actors it knows; and whether or not that actor has messages pending in its mail queue.

2. PROBLEM DEFINITION

The actor model captures the essential properties of a system of active objects.

Actors are formally defined in [2]. In the actor model, the universe consists of autonomous computational agents called *actors*. Each actor has a unique *mail address* which can be used to communicate with that actor. Communication between actors is asynchronous with unbounded delay but guaranteed delivery. Messages for an actor are buffered in a *mail queue*. Computation is message driven and each actor processes the messages it receives in its mail queue in the order they were queued. Processing a message involves executing a script which is called the *behavior* of the actor. In response to a message, an actor may create new actors, send messages to actors and possibly change its behavior as shown in Figure 1 (See [3]).

The essentials of actor systems that are important in our discussion about garbage collection are:

1. An actor which is processing a message can communicate with another actor it knows of.
2. Mail addresses can be communicated in messages.
3. Communication is asynchronous and buffered (the sender does not wait till the recipient is ready to receive a message).
4. Message delivery time is unbounded but all messages are eventually delivered. Note that this assumption is required to show that all garbage is eventually recognized (a liveness property).

Let $G = (V, E)$ be a graph, with V as the set of actors and an edge present from a to b if actor a has a reference to b. b is called a *forward acquaintance* of a and a is called an *inverse acquaintance* of b. This graph is called the *actor reference* graph. Note that the forward and the inverse acquaintance each define a binary relation on the set of actor names.

A subset of actors $\rho \subseteq V$ is called a *root* set and actors in this set are never considered garbage. As an example, in a module, the receptionist actors which are accessible from outside the system could be considered root. An actor which is either processing a message or has messages pending in its mail-queue is called an *unblocked* actor. An actor which is not unblocked is called a *blocked* actor. A blocked actor which is not connected by the recursive closure of the inverse acquaintance relation to an unblocked actor is called a *permanently blocked* actor.

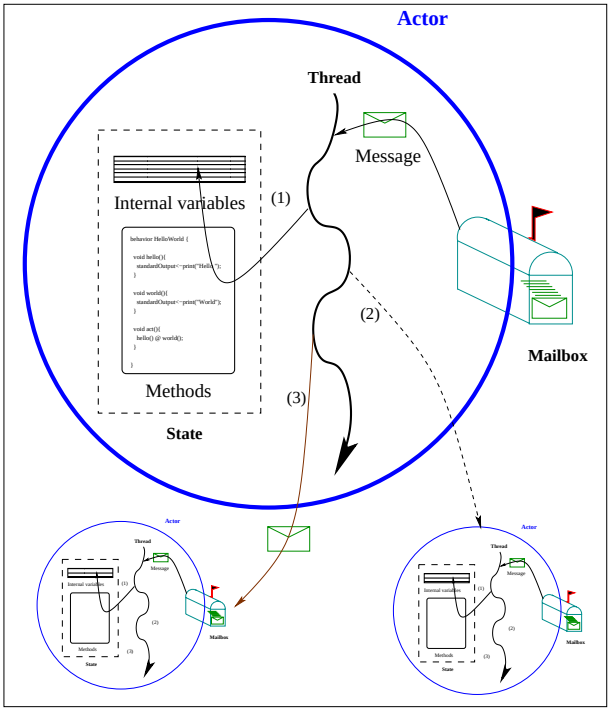

Figure 1: In response to a message, an actor can: (1) modify its local state, or (2) create new actors, or (3) send messages to acquaintances.

The set of *live* actors is defined inductively by the following rules:

DEFINITION 1. **Identifying live actors**

1. A root actor is live.
2. If an actor a is live, a forward acquaintance of a is live.
3. If an actor a is live, an inverse acquaintance of a which is not permanently blocked is live.

An actor which is not live is *garbage*. As an example application of the above rules, consider the actor reference graph shown on the top part of Figure 3. It can be seen the actors that are live are 1, 2, 3, 4, 5, 6 and 8.

The property of being garbage is a stable property: if an actor is garbage in a given state, it will remain garbage in any possible state derived from that state. For a detailed presentation of the differences in garbage collection of passive object and actor systems, the reader is referred to Vardhan[35].

3. RELATED WORK

3.1 Actor garbage collection

A formal definition of garbage actors was first given by Kafura *et al.*[18]. The basis of the algorithms presented in their work is: marking actors with three different colors, *viz.* black, gray and white, with black being the only color

which guarantees that an actor is live. The colors have the following meaning:

- Black: It has been shown that the actor is either a root, can receive a message from a root or can send a message to a root.
- Gray: The actor is not processing any messages and does not have any pending messages but can communicate with a root actor if it later receives a message.
- White: It has not been shown that the actor can communicate with a root actor.

Initially, roots are marked black and the rest are marked white. The color of actors is changed in accordance with a set of coloring rules until no further change is possible. Actors which do not get colored black are reclaimed as garbage. In a more recent paper, Kafura *et al.*[17] have described in detail a distributed version of their algorithm. Local garbage collection at different hosts is allowed to proceed independently. A distributed global collector is invoked for collecting garbage not recognizable by the local collectors. The collector uses previously available algorithms for taking consistent snapshots in a distributed system and for detecting termination.

A hierarchical distributed garbage collection algorithm is described by Venkatasubramanian *et al.*[36]. An approach similar to mark-sweep is followed, with specialized marking rules which are formulated according to the definition of garbage in actors. A snapshot of the global system is formed and messages in the network sent before an initial broadcast are accounted for by sending "bulldoze" messages across the network. The distributed system is partitioned into clusters which are organized hierarchically to avoid the bottleneck of computation and resource management.

A garbage collector for active objects in a massively parallel environment has been described by Kamada *et al.*[19]. In this work, all active objects are considered live. An interesting feature of their algorithm is the use of a centralized agent which handles many problems related to synchronization and detection of termination of various phases of the garbage collection.

Puaut [28, 29] presents an algorithm comprising of independent local collectors loosely coupled to a global collector. The global collector is a logically centralized service that maintains a graph which is a merge of subgraphs sent by the local garbage collectors.

Dickman [12] presents an interesting algorithm called the *Partition Merging algorithm* (PMA). A key idea of the algorithm is that all actors reachable from an unblocked actor (including the unblocked actor itself) have the same garbage status. The main action of PMA is to form Eulerian cycles[1] of actors having the same garbage status due to their being reachable from one or more unblocked actors. The use of Euler cycles enables the traversal to be completed in linear time.

3.2 Distributed garbage collection

A number of collectors have been developed for distributed object systems. Some of the algorithms based on reference counting or reference listing are given by Bevan [7], Watson and Watson [38], Piquer [27], Shapiro *et al.*[34] and Birrel *et al.*[8]. Rodriguez-Rivera *et al.*[31] propose an algorithm based on reference listing augmented with back tracing. A similar approach is followed by Maheshwari [26]. Rodriguez *et al.*[30] also suggest an algorithm based on reference listing with partial tracing in order to collect cyclic garbage. Tracing is initiated at an object suspected to be a part of a garbage cycle. Augusteijn [5] presents an algorithm based on an incremental three-color mark-sweep algorithm applied to a distributed object system. Juul *et al.*[15] propose another distributed version of the incremental mark-sweep algorithm. Ladin and Liskov [20] propose an algorithm that relies on a logically centralized global garbage detection service.

Hughes [14] describes an appealing algorithm where mark bits are replaced by timestamps. The key idea is that a garbage object's timestamp remains constant whereas a non-garbage object timestamp increases monotonically. Fessant *et al.*[22] present an algorithm based on Hughes' algorithm in a simplified form that makes fewer assumptions about the distributed system. Lang, Queinnec and Piquer [21] suggest combining reference counting and mark-sweep in order to perform garbage collection within groups.

Schelvis [32] proposes a comprehensive global garbage collection algorithm based on time-stamp packet distribution. Local garbage collection on different hosts proceeds independently while a global garbage detection strategy tries to reclaim the entrance nodes which have become garbage. For global garbage detection, packets are asynchronously and repeatedly sent to each remotely referenced object. The algorithm does not require any synchronization between different processors but is still able to collect all garbage.

Louboutin *et al.*[24, 25, 23] present an interesting algorithm which is able to collect all distributed garbage by tracking causal dependencies of relevant mutator events. Each remotely referenced object (called global root) maintains a log of timestamps of edge-creation or edge-deletion events received from adjacent global roots. The timestamps are propagated along the paths of the global root graph enabling each global root to construct causal histories of relevant events which ultimately identifies objects which have become garbage. The algorithm is reactive, incremental, scalable, does not require synchronization among processes, and is able to collect cyclic garbage.

4. GARBAGE COLLECTION FRAMEWORK

Without loss of generality we assume that there is a single root actor r in the actor-reference graph: if there are more than one root actors, we can add a hypothetical root actor which has references to the actual roots. Again without loss of generality we assume that the root actor is always unblocked. This is because whether or not the root is unblocked is only important for deciding liveness for an actor b which has the root in the recursive closure of the inverse acquaintance relation (by application of Rule 3 for identifying live actors as in Definition 1). Thus, we might decide that b is not permanently blocked when in fact it might have been considered permanently blocked if the root was blocked. The concern would then be that b should not be incorrectly classified as live. However, b must be in the recursive closure of the forward acquaintance relation from the root and hence by successive applications of Rule 2, will be considered live regardless of the application of Rule 3. Therefore, it makes

[1] An Euler cycle in a connected directed graph is a cycle that traverses each edge of the graph exactly once.

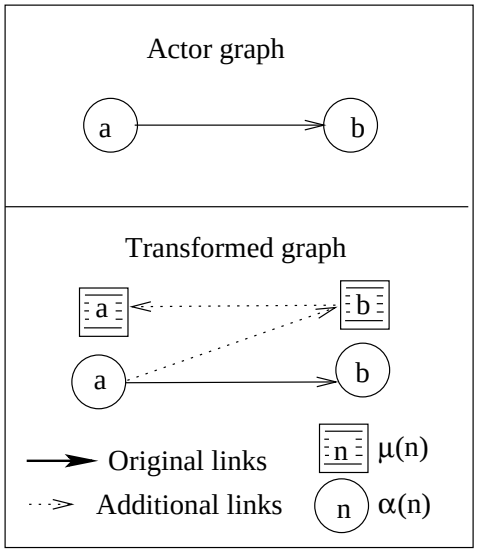

Figure 2: Transformation of an acquaintance in the actor-reference graph

no difference to the garbage status of any actor whether the root is blocked or unblocked.

Given the actor reference graph $G = (V, E)$ and a root actor $\rho \in V$, we define a transformation function $\tau : (G, \rho) \rightarrow (G', \rho')$ where $G' = (V', E')$ is another graph and $\rho' \in V'$. The nodes and edges of G' are constructed from the following rules:

RULES 1. **Transformation** $\tau : (G, \rho) \rightarrow (G', \rho')$ Let α and μ be bijective functions from actor names to labels such that $\text{Range}(\alpha) \cap \text{Range}(\mu) = \emptyset$.

1. The root object ρ' in G' is given by $\rho' = \mu(\rho)$.
2. For every actor named a in V, there are two corresponding nodes: $\alpha(a) \in A'$ and $\mu(a) \in M'$. $V' = A' \cup M'$.
3. If an actor a is unblocked, there is an edge from $\mu(a)$ to $\alpha(a)$ in G'.
4. If an actor a has a reference to an actor b, there is an edge from $\alpha(a)$ to both $\alpha(b)$ and $\mu(b)$; and an edge from $\mu(b)$ to $\mu(a)$. Figure 2 illustrates this

Now the problem of actor garbage collection can be solved by the following algorithm:

ALGORITHM 1.

1. *Obtain a snapshot $G = (V, E)$ of the actor reference graph. This can be done by any of the standard techniques for obtaining distributed snapshots, see for example [37, 10].*
2. *Apply the transformation to obtain $(G', \rho') = \tau(G, \rho)$ with $G' = (V', E')$ and $V' = A' \cup M'$.*
3. *Run any passive object garbage collection on G' with V' as the objects; E' as the edges defining the references; and root object ρ'. Let $V'_g \subset V'$ be the objects found as garbage on G'.*
4. *For all $v' \in (V'_g \cap A')$, actor $\alpha^{-1}(v')$ is declared as garbage.*

Before proving the correctness of this algorithm, we provide some intuition about the rules of transformation of the actor reference graph. The key property that makes actor garbage collection different from passive object garbage collection is that an unblocked actor can change the reference graph making itself reachable. The transformation is designed to produce an object reference graph that automatically captures this property of actors with respect to garbage collection. For an actor a, we can think of $\alpha(a)$ as the object corresponding to the actor itself and $\mu(a)$ as the object corresponding to its mail queue. If a has a reference to another actor b, the reference from $\alpha(a)$ to $\alpha(b)$ simply translates the reference from a to b. The references from $\alpha(a)$ to $\mu(b)$ and $\mu(b)$ to $\mu(a)$ are present to account for the possibility that a could send a message to b sending its own address and making it reachable from b. But notice that since the rules make only $\mu(a)$ (and not $\alpha(a)$) reachable from $\mu(b)$, $\alpha(a)$ does not "benefit" from these additional references unless a is unblocked (and hence $\alpha(a)$ has a reference from $\mu(a)$). In this manner all the relevant information of actors a and b with respect to garbage collection is captured in the transformed graph.

In a distributed system, the actor reference graph may be spread across different hosts and may be changing dynamically. Further, there might be messages in transit which have not yet arrived in the destination actor's mail queue. These problems are resolved by obtaining a global snapshot of the actor reference graph. Since there is no omniscient observer in a distributed system, a global snapshot is usually constructed by taking a consistent cut of the global state as described in [37, 10].

We illustrate the transformation in Figures 3 and 4. For actor names $i = \{1, 6, 10, 12\}$ which are unblocked, there is an edge from $\mu(i)$ to $\alpha(i)$. Looking at this graph we can see that a garbage collector for passive objects would regard $\alpha(1)$, $\alpha(2)$, $\alpha(3)$, $\alpha(4)$, $\alpha(5)$, $\alpha(6)$ and $\alpha(8)$ as live and all others objects in A' as garbage. A look at the original actor-reference graph shows that it is exactly actors 1, 2, 3, 4, 5, 6 and 8 that are live. Of special interest is $\alpha(6)$ in the transformed graph. Because $\alpha(6)$ has a reference from $\mu(6)$ which is reachable from $\mu(1)$ (the root), it is correctly identified as being live. The reader can also note that, although $\mu(7)$ is reachable in the transformed graph, $\alpha(7)$ is not. By step 4 of Algorithm 1, it is $\alpha(7)$ that is used for deciding garbage status of actor 7 and hence 7 is correctly identified as garbage.

LEMMA 1. *For any actor a, if $\alpha(a)$ is found live by a passive object garbage collector in the transformed graph G', $\mu(a)$ will also be live.*

PROOF. In the shortest path from the root to $\alpha(a)$, let b' the object before $\alpha(a)$ (note that $\alpha(a)$ cannot be the root in G'). By the rules of the transformation, b' is either $\mu(a)$ or $\alpha(b)$ for some other actor b. If b' is $\mu(a)$ then $\mu(a)$ is reachable from the root and is live. For the other case, notice that by the rules of the transformation, $\alpha(b)$ will have a reference to $\mu(a)$ making it also live. □

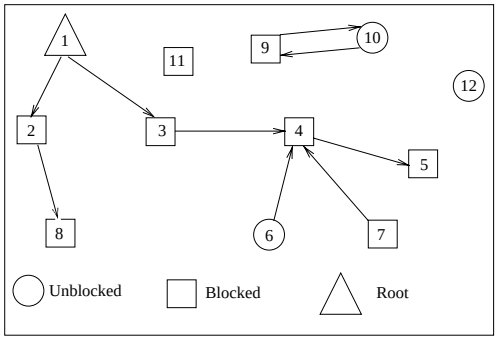

Figure 3: Original actor-reference graph

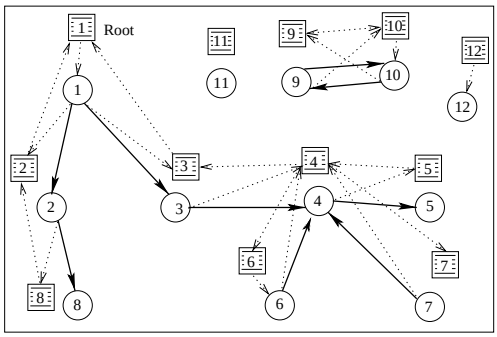

Figure 4: Transformed object graph

THEOREM 1. *An actor a is live in $G \Rightarrow$ object $\alpha(a)$ is live in G'*

PROOF. Let $P(a)$ be the property of actor a that object $\alpha(a)$ is live in G'. We have to show that $P(a)$ holds for any actor a which is live. We use rule induction on the set of live actors as given by Definition 1. The base case is for the root actor ρ. By the definition of the transformation, $\mu(\rho)$ is the root object in G'. Further since the root actor can be assumed to unblocked, $\alpha(\rho)$ is reachable from $\mu(\rho)$ implying that $\alpha(\rho)$ is live in G'. Hence, $P(\rho)$ holds.

Now we have to show that for all instances of the rules, if $P(a)$ holds for a given live actor a, it also holds for the actor derived to be live because of that rule.

Rule 2 states that if an actor a is live, then a forward acquaintance of a [call it b] is live. By the rules of the transformation, $\alpha(a)$ has a reference to $\alpha(b)$. Hence, if $P(a)$ holds, $\alpha(b)$ is also live in G' which means that $P(b)$ holds.

Rule 3 states that if an actor a is live, then an inverse acquaintance of a which is not permanently blocked [call it b] is live. Let $d_1, d_2 \ldots d_n$ be the sequence of actors with d_1 being an unblocked actor and $d_n = b$ such that d_{i+1} is a forward acquaintance of d_i for $i = 1 \ldots n-1$. There has to be some such sequence since b is not permanently blocked (note that n could be 1). If there are more than one such sequences, we pick one with the smallest n. By the rules of the transformation, $\mu(d_1)$ has a reference to $\alpha(d_1)$; $\mu(d_{i+1})$ has a reference to $\mu(d_i)$ and $\alpha(d_i)$ has a reference to $\alpha(d_{i+1})$ for $i = 1 \ldots n-1$. Moreover, $\mu(a)$ has a reference to $\mu(b)$. Following these references as shown in Figure 5, we can see that $\alpha(b)$ is reachable from $\mu(a)$. If $P(a)$ holds then $\alpha(a)$ is live in G' and using Lemma 1 we can conclude that $\mu(a)$ is live. Hence $\alpha(b)$ is live in G' demonstrating that $P(b)$ holds. □

THEOREM 2. *An object $a' \in A'$ is live in $G' \Rightarrow$ actor $\alpha^{-1}(a')$ is live in G*

PROOF. We use induction on the length of the shortest path from ρ' to a'. Since $\rho' \notin A'$, the length of the shortest path from ρ' has to be at least 1. For this base case, the only possibility is that $a' = \alpha(\rho)$ since by definition, all the other references that $\mu(\rho)$ has, can only be to objects in M'. The claim holds trivially for this base case.

Assuming the induction hypothesis for lengths up to k, consider an object $a' \in A'$ found to be live in G' with shortest path length $k+1$. Let actor a be $\alpha^{-1}(a')$. Along the shortest path from ρ', let the sequence of objects be $\rho', d'_1, d'_2, \ldots, d'_k, a'$. Then there are two possibilities:

$d'_k \in A'$. The induction hypothesis applies for d'_k, hence actor $\alpha^{-1}(d'_k)$ must be live. But then by the rules of the transformation, $\alpha^{-1}(d'_k)$ has a reference for a, hence a is also live.

$d'_k = \mu(a)$. Since this means that $\mu(a)$ has a reference to $\alpha(a)$, by the rules of the transformation, this implies that actor a is unblocked. Consider the following cases:

- At least one object in $\{d'_1, d'_2 \ldots d'_k\}$ is in A': Let d'_j be the object with largest j such that $d'_j \in A'$. This implies $d'_i \in M' \quad \forall i = j+1 \ldots k$. By the induction hypothesis, $\alpha^{-1}(d'_j)$ is live in G. By the rules of the transformation, actor $\mu^{-1}(d'_{j+1})$ is a forward acquaintance of $\alpha^{-1}(d'_j)$ and hence live. Further $\mu^{-1}(d'_{j+1})$ has a in the reflexive closure of the inverse acquaintance relation. Since a is unblocked, $\mu^{-1}(d'_{j+1})$ is not permanently blocked and being an inverse acquaintance of a live actor ($\alpha^{-1}(d'_j)$) is live by Rule 3. Extending this argument successively to $\mu^{-1}(d'_{j+2})$, $\mu^{-1}(d'_{j+3})$, $\ldots$ $\mu^{-1}(d'_k)$, a we can see that all of these are live as shown in Figure 6.
- Otherwise: All of $\{d'_1, d'_2 \ldots d'_k\}$ are in M'. Using the rules of transformation, it is easy to see that $\mu^{-1}(d'_1)$ has a in the reflexive closure of the inverse acquaintance relation. Since a is unblocked, $\mu^{-1}(d'_1)$ is not permanently blocked and being an inverse acquaintance of a live actor (the root) is live by Rule 3. Again, by extending this argument successively to $\mu^{-1}(d'_2)$, $\mu^{-1}(d'_3)$, $\ldots$ $\mu^{-1}(d'_k)$, a we see that all of these are live.

□

THEOREM 3.

1. *Any actor a declared to be garbage by Algorithm 1 is indeed garbage according to the formal definition of garbage actors given in Definition 1*
2. *Any actor a which can be found to be garbage based on Definition 1 will eventually be declared as garbage by Algorithm 1.*

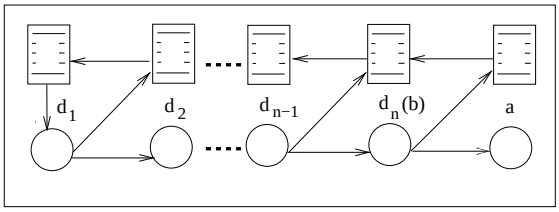

Figure 5: Illustration for proof of Theorem 1

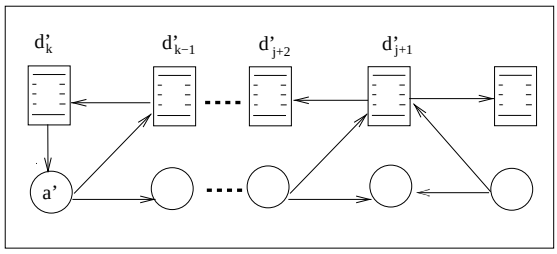

Figure 6: Illustration for proof of Theorem 2

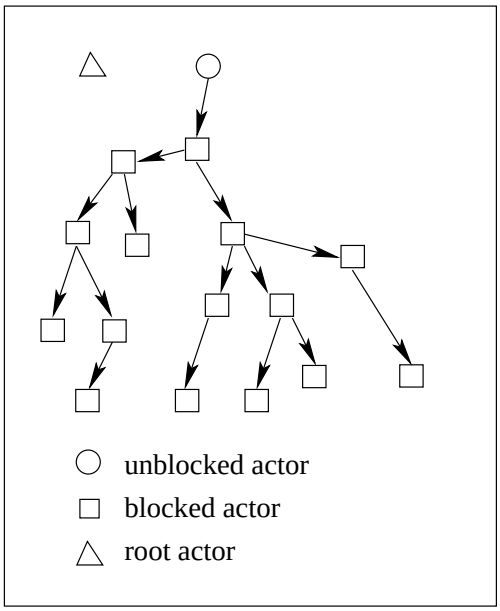

Figure 7: A reference graph in which maintaining inverse acquaintances is advantageous.

PROOF.

1. Algorithm 1 declares an actor a to be garbage, if in some snapshot G of the actor reference graph, $\alpha(a)$ is garbage in the transformed graph G'. By the contrapositive of Theorem 1, a has to be garbage in G according to Definition 1. Since an actor once garbage can never become live again, a is indeed garbage in the actor reference graph.

2. If an actor a is in fact garbage, since all messages are eventually delivered, some snapshot of the actor reference graph is bound to capture it as being garbage according to Definition 1. By the contrapositive of Theorem 2, $\alpha(a)$ will be found to be garbage in the transformed graph G'. Hence a will be correctly identified as garbage by Algorithm 1.

□

5. COST OF TRANSFORMATION AND OPTIMIZATIONS

For an actor a, the extra information added in the transformed graph is the addition of another object; the reference between $\mu(a)$ and other objects; and the reference between $\mu(a)$ and $\alpha(a)$ for an unblocked actor. At first sight this might appear to be excessive overhead. However, in practice, we can encode both $\mu(a)$ and $\alpha(a)$ in a single object. The link between the $\mu(a)$ and $\alpha(a)$ can be represented by a single bit in the state of this object, with the passive object collector modified to recognize this bit as a reference. In a mark-sweep like scheme, separate mark bits can be kept for both the $\mu(a)$ and $\alpha(a)$ in the state of the object itself. We do have to maintain inverse acquaintances in order for $\mu(a)$ to be able to refer to $\mu(b)$ for any inverse acquaintance b.

The main additional cost in transforming the graph is building of inverse acquaintances for all actors. The inverse acquaintances can be established at the time of garbage collection or can be maintained incrementally as acquaintances are created. Maintaining inverse acquaintances presents a trade off. There are some reference graphs in which maintaining inverse acquaintances would be an unnecessary overhead. However, in certain reference graphs it can be advantageous. Consider Figure 7, which shows a reference graph with an unblocked actor having a large number of actors in the reflexive closure of the forward acquaintance relation. The root actor does not have any forward or inverse acquaintances. Any algorithm which does not keep inverse acquaintances would have to trace all the actors in the reflexive closure in order to make sure there is no actor through which a message can be passed to the root actor. On the other hand, if inverse acquaintances are maintained it can be easily seen without tracing the entire reachability set that the set is garbage.

6. AN EXAMPLE INSTANTIATION

Our approach towards garbage collection of actors gives us a family of algorithms which are parameterized by the choice of the passive object garbage collector run on the transformed graph. If we choose a simple mark-sweep as the passive object garbage collector, the resulting algorithm that we get bears a close similarity to a known algorithm for garbage collection of actors given by Venkatasubramanian *et al.*[36]. Their algorithm starts by marking root actors as *touched*. For each *touched* actor, all forward acquaintances and non-blocked inverse acquaintances are marked *touched*. Blocked inverse acquaintances are marked as *suspended*. Blocked inverse acquaintances of *suspended* actors are marked *suspended* but unblocked inverse acquaintances are marked *touched*. The process continues till the marking color of no actor can be changed. At that point, all *touched* actors are considered as live and others as garbage.

Our mark-sweep on the transformed object-reference graph proceeds in a manner identical to the algorithm given by

Venkatasubramanian *et al.*. When their algorithm proceeds to mark an actor *a touched*, our algorithm proceeds to mark both $\alpha(a)$ and $\mu(a)$. When an actor is marked *suspended* in their algorithm, $\mu(a)$ but not $\alpha(a)$ is marked in our algorithm.

7. DISCUSSION

We have implemented a garbage collector for actors based on our framework in the Actor Foundry [1], which is an actor system written in Java. The run-time environment of the Actor Foundry consists of one or more Java Virtual Machines running on possibly different hosts. The passive garbage collector which runs on the transformed graph uses the Schelvis [32] algorithm based on time-stamp packet distribution. Local garbage collection is allowed to proceed independently on each host and a global garbage collection service collects garbage which cannot be recognized on the basis of local information alone. For details of the implementation, the reader is referred to [35].

In systems which have actors as well as passive objects, our framework would be particularly useful since it would offer a common approach for garbage collection. By extending the transformation described in this paper to references which go from passive objects to actors and *vice versa*, a passive garbage collector on the transformed graph could be used to recognize both garbage actors and garbage objects in the original graph. This would avoid running two separate garbage collectors for actors and passive objects respectively.

The cost of using the GC transformation we have presented is not substantial. One requirement for the transformation is the knowledge of inverse acquaintances of all actors. Although such a requirement may be considered to be an overhead, in certain cases maintaining inverse acquaintances can reduce the effort required for garbage collection. Moreover, our transformation provides an elegant method to integrate garbage collection of active and passive objects in systems that support both kinds of objects.

8. REFERENCES

[1] The Actor Foundry. http://www-osl.cs.uiuc.edu/foundry.

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

[3] G. Agha, N. Jamali, and C. Varela. Agent naming and coordination: Actor based models and infrastructures. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, chapter 9, pages 225–246. Springer-Verlag, Mar. 2001.

[4] J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[5] L. Augusteijn. Garbage collection in a distributed environment. In de Bakker et al. [11], pages 75–93.

[6] Y. Bekkers and J. Cohen, editors. *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 Sept. 1992. Springer-Verlag.

[7] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187. Springer-Verlag, June 1987.

[8] A. Birrell, D. Evers, G. Nelson, S. Owicki, and E. Wobber. Distributed garbage collection for network objects. Technical Report 116, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, Dec. 1993.

[9] J.-P. Briot. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In S. Cook, editor, *Proceedings ECOOP'89*, pages 109–129, Nottingham, 10-14 1989. Cambridge University Press.

[10] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[11] J. W. de Bakker, L. Nijman, and P. C. Treleaven, editors. *PARLE'87 Parallel Architectures and Languages Europe*, volume 258/259 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 1987. Springer-Verlag.

[12] P. Dickman. Incremental, distributed orphan detection and actor garbage collection using graph partitioning and euler cycles. In O. Babaoglu and K. Marzullo, editors, *Tenth International Workshop on Distributed Algorithms WDAG'96*, volume 1151 of *Lecture Notes in Computer Science*, Bologna, Oct. 1996. Springer-Verlag.

[13] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, Jan. 21-24 1996. ACM.

[14] R. J. M. Hughes. A distributed garbage collection algorithm. In J.-P. Jouannaud, editor, *Record of the 1985 Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 256–272, Nancy, France, Sept. 1985. Springer-Verlag.

[15] N.-C. Juul and E. Jul. Comprehensive and robust garbage collection in a distributed system. In Bekkers and Cohen [6].

[16] D. Kafura, M. Mukherji, and G. Lavender. A class library for concurrent programming in C++ using actors, 1993.

[17] D. Kafura, M. Mukherji, and D. Washabaugh. Concurrent and distributed garbage collection of active objects. *IEEE Transactions on Parallel and Distributed Systems*, 6(4), Apr. 1995.

[18] D. Kafura, D. Washabaugh, and J. Nelson. Garbage collection of actors. In N. Meyrowitz, editor, *OOPSLA'90 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 25(10) of *ACM SIGPLAN Notices*, pages 126–134, Ottawa, Ontario, Oct. 1990. ACM Press.

[19] T. Kamada, S. Matsuoka, and A. Yonezawa. Efficient parallel global garbage collection on massively parallel computers. In E. Moss, P. R. Wilson, and B. Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, Oct. 1993.

[20] R. Ladin and B. Liskov. Garbage collection of a

distributed heap. In *International Conference on Distributed Computing Systems*, Yokohama, June 1992.

[21] B. Lang, C. Quenniac, and J. Piquer. Garbage collecting the world. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 39–50. ACM Press, Jan. 1992.

[22] F. Le Fessant, I. Piumarta, and M. Shapiro. An implementation for complete asynchronous distributed garbage collection. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[23] S. Louboutin and V. Cahill. A lazy log-keeping mechanism for comprehensive global garbage detection on Amadeus. In *OOIS (Object-Oriented Information Systems) '95*, pages 118–132, London, Dec. 1995. Springer-Verlag. Technical report TCD–CS–95–11.

[24] S. R. Louboutin. *A Reactive Approach to Comprehensive Global Garbage Detection*. PhD thesis, Trinity College, Dublin, 1998. In preparation.

[25] S. R. Louboutin and V. Cahill. Comprehensive distributed garbage collection by tracking causal dependencies of relevant mutator events. In *Proceedings of ICDCS'97 International Conference on Distributed Computing Systems*. IEEE Press, 1997.

[26] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by back tracing. In *Proceedings of PODC'97 Principles of Distributed Computing*, 1997.

[27] J. M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In Aarts et al., editors, *PARLE'91 Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1991.

[28] I. Puaut. Distributed garbage collection of active objects with no global synchronisation. In Bekkers and Cohen [6].

[29] I. Puaut. A distributed garbage collector for active objects. In *PARLE'94 Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science. Springer-Verlag, 1994. Also INRIA UCIS-DIFUSION RR 2134.

[30] H. C. C. D. Rodrigues and R. E. Jones. Cyclic distributed garbage collection with group merger. In E. Jul, editor, *Proceedings of 12th European Conference on Object-Oriented Programming, ECOOP98*, Lecture Notes in Computer Science, pages 249–273, Brussels, July 1998. Springer-Verlag. Also UKC Technical report 17–97, December 1997.

[31] G. Rodriguez-Riviera and V. Russo. Cyclic distributed garbage collection without global synchronization in CORBA. In P. Dickman and P. R. Wilson, editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, Oct. 1997.

[32] M. Schelvis. Incremental distribution of timestamp packets — a new approach to distributed garbage collection. *ACM SIGPLAN Notices*, 24(10):37–48, 1989.

[33] P. Sewell and P. Wojciechowski. Nomadic Pict: Language and infrastructure design for mobile agents, 2000.

[34] M. Shapiro, P. Dickman, and D. Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapports de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, Nov. 1992. Also available as Broadcast Technical Report 1.

[35] A. Vardhan. Distributed garbage collection of active objects: A transformation and its applications to Java programming. Master's thesis, University of Illinois at Urbana Champaign, October 1998.

[36] N. Venkatasubramanian, G. Agha, and C. Talcott. Scalable distributed garbage collection for systems of active objects. In Bekkers and Cohen [6], pages 134–147.

[37] N. Venkatasubramanian and C. L. Talcott. Reasoning about meta level activities in open distributed systems. In *Symposium on Principles of Distributed Computing*, pages 144–152, 1995.

[38] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In de Bakker et al. [11], pages 432–443.

[39] A. Yonezawa, J. R. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proceedings of the 1986 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, pages 258–268. ACM Press, 1986.