Actor Garbage Collection Using Vertex-Preserving Actor-to-Object Graph Transformations

Wei-Jen Wang[1], Carlos Varela[2], Fu-Hau Hsu[1]*, and Cheng-Hsien Tang[1]

[1] Department of Computer Science and Information Engineering
National Central University, Taiwan
[2] Department of Computer Science
Rensselaer Polytechnic Institute, USA

Abstract. Large-scale distributed computing applications require concurrent programming models that support modular and compositional software development. The actor model supports the development of independent software components with its asynchronous message-passing communication and state encapsulation properties. Automatic actor garbage collection is necessary for high-level actor-oriented programming, but identifying live actors is not as intuitive and easy as identifying live passive objects in a reference graph. However, a transformation method can turn an actor reference graph into a passive object reference graph, which enables the use of passive object garbage collection algorithms and simplifies the problem of actor garbage collection. In this paper, we formally define *potential communication* by introducing two binary relations - the *may-talk-to* and the *may-transitively-talk-to* relations, which are then used to define the set of *live actors*. We also devise two vertex-preserving transformation methods to transform an actor reference graph into a passive object reference graph. We provide correctness proofs for the proposed algorithms. The experimental results also show that the proposed algorithms are efficient.

Key words: Garbage collection, actors, active objects, program transformation

1 Introduction

Parallel and distributed computing applications are working on increasingly larger data sets and require more parallelism to better exploit new hardware such as multi-core architectures and graphical processing units. These applications demand concurrent programming models that support modular and compositional software development. The actor model of computation [1–3] can be used to reason about and to build such massively parallel and distributed computing systems. The fundamental computing unit of actor systems is a reactive entity called the *actor*, which encapsulates a thread of control along with its internal state. Communication of actors is through asynchronous message passing,

in which message sending is non-blocking and message reception is unordered (non-FIFO).

High-level actor-oriented programming languages [4] support dynamic actor creation and actor reference passing, such as SALSA [3], Scala [5], E [6] and Erlang [7]. Introducing automatic actor garbage collection [8–10] to actor-oriented programming languages simplifies the problem of dynamic lifetime management of actors because the computing resources occupied by actor garbage can be reclaimed without manual intervention. Automatic actor garbage collection can reduce programmers' efforts on their sometimes error-prone manual lifetime management of actors, and can also let them focus on developing application functionality. Therefore automatic actor garbage collection is necessary for high-level actor-oriented programming.

Actor Garbage Problem

The widely used definition of live actors is described in [8]. Conceptually, the set of live actors consists of the set of root actors and the set of actors which can potentially communicate with the root set of actors. The set of actor garbage is then defined as the complement set of live actors. However, the term "potentially communicate with" is too abstract to make an operational definition. [10] proposes a more operational definition of actor garbage.

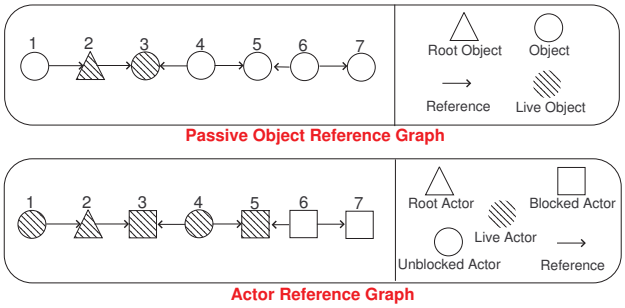

Fig. 1. Objects 2 and 3 are live, while Actors 1 to 5 are live.

Both the passive object garbage collection problem and the actor garbage collection problem can be expressed as directed graph problems, consisting of directed edges (references) and vertices (objects or actors). The major difference resides in the state of each vertex — a non-root object has only one state; a non-root actor can either be *unblocked* or *blocked* and its state changes dynamically. An actor is unblocked if it is processing a message or has messages in its message box; it is blocked otherwise. Consider the example in Figure 1. It illustrates that

a passive object reference graph and an actor reference graph can have different set of live objects/actors even given the same set of vertices and references. The upper half of Figure 1 shows a passive object reference graph, where Objects 2 and 3 are live. The lower half of Figure 1 shows a similar actor reference graph, where Actors 1 to 5 are live. Actor 1 is live because it can send a message to the set of root actors. Actors 3 to 5 are live because Actor 2 and Actor 4 can send their references to Actor 3 and Actor 5, and change them to the unblocked state respectively. As a result, Actors 3 to 5 can become unblocked and can transitively send messages to the root set of actors.

Transformation Technique

An actor garbage collection algorithm is not as intuitive as a passive object collection algorithm [11,12] because it has more restrictions. However, Vardhan and Agha [9] have pointed out that an actor reference graph can be transformed into a passive object reference graph, which enables the use of passive object garbage collection algorithms. The problem of the transformed reference graph is that it produces at least three times as many references and about twice as many vertices as the original actor reference graph. Figure 2 shows an object reference graph which is transformed from the actor reference graph in Figure 1 using the Vardhan-Agha transformation method. The transformed reference graph consists of 14 vertices and 21 edges.

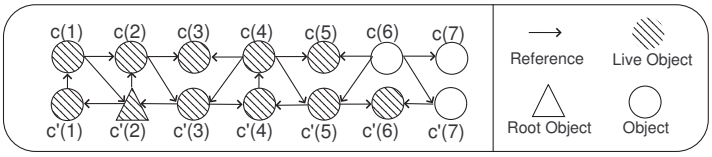

Fig. 2. A transformed reference graph from the actor reference graph in Figure 1 using the Vardhan-Agha transformation method. Objects $c(1)$ to $c(5)$ are live, implying Actors 1 to 5 are live.

Contributions

In this paper, we formally define *potential communication* by introducing two binary relations — the *may-talk-to* and the *may-transitively-talk-to* relations, which are then used to define the set of live actors. We devise two vertex-preserving transformation methods, each of which can transform an actor reference graph into a passive object reference graph. Compared to the Vardhan-Agha transformation method, both of our methods only require adding new edges for

transformation. The experimental results also show that our algorithms are more efficient than the Vardhan-Agha transformation method.

Outline of This Paper

The rest of the paper is organized as follows: Section 2 briefly introduced related actor garbage collection work. Section 3 defines garbage. Section 4 provides two novel vertex-preserving transformation methods, and proves that actor garbage collection is equivalent to passive object garbage collection. Section 5 explains how we implement the proposed algorithms, and compare them to the Vardhan-Agha algorithm. Section 6 presents conclusions.

2 Related Work

At the beginning stage of the actor model (the 1970s and the early 1980s), state encapsulation is not a requisite. Processes (or processors) which use asynchronous messages for communication are usually considered as actors, even though processes can have references to remote objects. Halstead's algorithm [13] deals with garbage collection on a set of connected processors, each of which is viewed as a root and maintains references to objects. Based on the same assumption of memory sharing and Baker's garbage collection algorithm [14], Lieberman and Hewitt [15] proposed a concurrent generational garbage collection algorithm which spends proportionately less effort reclaiming objects with longer lifetimes, and makes different storage for objects according to their age.

In the middle 1980s, Agha emphasized the importance of state encapsulation in the actor model [1] because it is an essential feature for modular software development. As a result, the change of the actor model leads to the need of new actor marking strategies because traditional passive object garbage collection strategies cannot be directly reused. Kafura et al. [8, 16, 17] proposed the Push-Pull algorithm and the Is-Black algorithm. Dickman proposed the *partition merging* algorithm [18] which treats all unblocked actors as potential roots, divides actors into several large partitions, and then verifies each partition from the root actors using an Euler cycle traversal algorithm. Vardhan and Agha proposed a problem transformation technique to do actor garbage collection which is described in Section 1 and [9].

3 Definition of Garbage

In this section, we formally define the passive object garbage collection problem using the *transitive reachability* relation $\rightsquigarrow$. We then propose a new model to define the actor garbage collection problem using the *may-talk-to* $\leftrightsquigarrow$ and the may-transitively-talk-to $\leftrightsquigarrow^*$ relations. The definitions will be used in Section 4 to prove equivalence of actor garbage collection and passive object garbage collection.

3.1 Garbage in Passive Object Systems

The essential concept of passive object garbage is based on the idea of the possibility of object manipulation. *Root* objects are objects that can be directly accessed by the thread of control. *Live* objects are those transitively reachable from the root objects by following references, while *garbage* objects are those who are not live. The problem of passive object garbage collection can be represented as a graph problem. To concisely describe the problem, we introduce *transitive reachability* $\rightsquigarrow$. The transitive reachability relation is *reflective* ($a \rightsquigarrow a$) and *transitive* ($(a \rightsquigarrow b) \wedge (b \rightsquigarrow c) \Rightarrow (a \rightsquigarrow c)$). Then we use it to define the passive object garbage collection problem.

Definition 1. *Transitive reachability* [3].
Object (or actor) o_q is transitively reachable from o_p, denoted by

$$o_p \rightsquigarrow o_q,$$

if and only if $o_p = o_q \vee (\exists o_u : \overline{o_p o_u} \wedge o_u \rightsquigarrow o_q)$.
Otherwise, we say $o_p \not\rightsquigarrow o_q$.

Definition 2. *Live passive objects.*
Given a passive object reference graph $G = \langle V, E \rangle$, where V represents objects and E represents references, let R represent roots such that $R \subseteq V$: The problem of passive object garbage collection is to find the set of live objects, $Live_{object}(G, R)$, where

$$Live_{object}(G, R) \equiv \{o_{live} \mid \exists o_{root} : (o_{root} \in R \wedge o_{live} \in V \wedge o_{root} \rightsquigarrow o_{live})\}$$

3.2 Garbage in Actor Systems

The definition of actor garbage is defined as having the ability to communicate with any of the *root actors*, where root actors are I/O services or public services such as web services and databases. We assume that *every actor/object has a reference to itself*, which is not necessarily true in the actor model [4]. The widely used definition of live actors proposed by Kafura et al. [8] is based on the possibility of message reception from or message delivery to the root actors — a *live* actor is one which can either receive messages from the root actors or send messages to the root actors. The original definition of live actors is denotational because it uses the concept of "potential" communication which must be considered along with possible state transitions from the present system configuration (system state). To make it more operational, the state of an actor (unblocked or blocked) and the referential relationship of actors must be used instead, as follows:

[3] $\overline{ab}$ is defined as a reference (or a directed edge) from a to b.
[4] The assumption that every actor/object has a reference to itself is used in most programming languages.

Definition 3. *Potential communication from a_p to a_q.*
Let the current system configuration be S. Potential communication from Actor a_p to Actor a_q (or message reception of a_q from a_p) is defined as:

$$\exists S_{future} : (a_p \text{ is unblocked} \wedge a_p \rightsquigarrow a_q \text{ at } S_{future}) \wedge (S \rightarrow^* S_{future}).$$

Definition 3 says that there exists potential communication from Actor a_p to Actor a_q if and only if both a_p will become unblocked and $a_p \rightsquigarrow a_q$ will become true at a future system configuration. In other words, the unblocked actor a_p can send a message to a_q along the path from a_p to a_q at a future system configuration.

Now, consider two actors, a_p and a_q. If they are both transitively reachable from an unblocked actor or a root actor, namely a_{mid}, message delivery from Actor a_p to Actor a_q (or from a_q to a_p) is possible. The reason is that there exists a sequence of state transitions such that a_{mid} transitively makes a_p unblocked and transitively creates a directional path to a_q. As a result, $a_p \rightsquigarrow a_q$ is possible. The relationship of a_p and a_q can be expressed by the *may-talk-to* relation, defined as $\leftrightsquigarrow$ (Definition 4). It is also possible that a message can be delivered from a_p to another new actor a_r if $(a_p \leftrightsquigarrow a_q \wedge a_q \leftrightsquigarrow a_r)$ because the unblocked actors can create a path to connect a_p and a_r. The generalized idea of the *may-transitively-talk-to* relation, $\leftrightsquigarrow^*$, is shown in Definition 5 to represent potential communication.

Definition 4. *May-talk-to $\leftrightsquigarrow$.*
Given an actor reference graph $G = \langle V, E\rangle$ and $\{a_p, a_q\} \subseteq V$, where V represents actors and E represents references, let R represent roots and U represent unblocked actors such that $R, U \subseteq V$, then:

$$a_p \leftrightsquigarrow a_q \iff \exists a_u : a_u \in (U \cup R) \wedge a_u \rightsquigarrow a_p \wedge a_u \rightsquigarrow a_q.$$

We call $\leftrightsquigarrow$ the may-talk-to relation.

Definition 5. *May-transitively-talk-to $\leftrightsquigarrow^*$.*
Following Definition 4,

$$a_p \leftrightsquigarrow^* a_q \iff a_p \leftrightsquigarrow a_q \vee \exists a_{mid} : (a_p \leftrightsquigarrow a_{mid} \wedge a_{mid} \leftrightsquigarrow^* a_q).$$

We call $\leftrightsquigarrow^*$ the may-transitively-talk-to relation.

Notice that the $\leftrightsquigarrow$ and $\leftrightsquigarrow^*$ relations are constrained by the set of unblocked actors and root actors of the current system configuration. By using the $\leftrightsquigarrow^*$ relation, the definition of the set of live actors can be concisely rewritten:

Definition 6. *Live actors.*
Given an actor reference graph $G = \langle V, E\rangle$, where V represents actors and E represents references, let R represent roots and U represent unblocked actors such that $R, U \subseteq V$. The problem of actor garbage collection is to find the set of live actors $Live_{actor}(G, R, U)$, where

$$Live_{actor}(G, R, U) \equiv \{a_{live} \mid \exists a_{root} : (a_{root} \in R \wedge a_{live} \in V \wedge a_{root} \leftrightsquigarrow^* a_{live})\}$$

4 Equivalence of Actor Garbage Collection and Passive Object Garbage Collection

This section presents two novel transformation methods that demonstrate the equivalence of object and actor garbage collection.

Transformation from Passive Object Garbage Collection to Actor Garbage Collection

Transformation from passive object garbage collection to actor garbage collection is easier because the passive object garbage collection problem is a sub-problem of actor garbage collection. Let the passive object reference graph be $G = \langle V, E\rangle$ and the set of roots be R. Let the transformed actor reference graph be $G' = \langle V', E'\rangle$, the set of roots be R', and U' be the set of unblocked actors. The problem of passive object garbage collection can be transformed into the problem of actor garbage collection by assigning $V' = V$, $E' = E$, $R' = R$ and $U' = \emptyset$. Then for any two objects o_r and o_q, we get $(o_r \rightsquigarrow o_q \wedge o_r \in R) \iff (o_r \rightsquigarrow o_r \wedge o_r \rightsquigarrow o_q \wedge o_r \in R) \iff (o_r \leftrightsquigarrow^* o_q \wedge o_r \in R)$. Therefore the set $Live_{object}(G, R) = Live_{actor}(G', R', U')$.

Transformation from Actor Garbage Collection to Passive Object Garbage Collection

Now, consider the backward transformation. Let the actor reference graph be $G = \langle V, E\rangle$, R be the roots and U be the unblocked actors. If there exist $G' = \langle V', E'\rangle$ and R' such that $Live_{actor}(G, R, U) = Live_{object}(G', R')$, we say the actor garbage collection problem can be transformed into the passive object garbage collection problem.

Transformation by Direct Back-Pointers to Unblocked Actors The direct back-pointer transformation method can transform actor garbage collection into passive object garbage collection by making $E' = E \cup \{\overline{a_q a_u} \mid a_u \in (U \cup R) \wedge a_u \rightsquigarrow a_q\}$. Figure 3 shows what the actor reference graph in Figure 1 transforms into. Notice that we use the term *back-pointers* to describe the newly added references and to avoid ambiguity with the term *references*. Theorem 1 shows that the direct back-pointer transformation method is correct.

Theorem 1. *Direct back-pointer transformation.*
Let the actor reference graph be $G = \langle V, E\rangle$, R be the set of roots, and U be the set of unblocked actors. Let $E' = E \cup \{\overline{a_q a_u} \mid a_u \in (U \cup R) \wedge a_u \rightsquigarrow a_q\}$, and $G' = \langle V, E'\rangle$ and $R' = R$.

$$Live_{actor}(G, R, U) = Live_{object}(G', R')$$

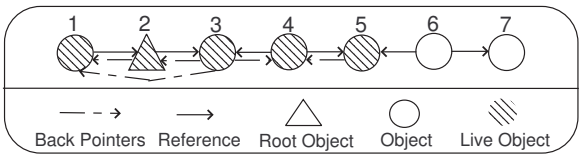

Fig. 3. An example of transformation by direct back-pointers to unblocked actors.

Proof. Let $Live_{actor}(G,R,U) = \{a_{live} \mid \exists a_{root} : (a_{root} \in R \wedge a_{live} \in V \wedge a_{root} \leftrightsquigarrow^* a_{live})\}$, and $Live_{object}(G',R') = \{o_{live} \mid \exists o_{root} : (o_{root} \in R' \wedge o_{live} \in V \wedge o_{root} \rightsquigarrow o_{live})\}$.
Now, consider the first case, $Live_{actor}(G,R,U) \subseteq Live_{object}(G',R')$. Let a_r and a_l be actors and $a_r \in R \wedge a_l \in V$. Then in G:
$a_r \leftrightsquigarrow^* a_l \Longrightarrow$
$\exists a_{mid,1}, a_{mid,2}, ..., a_{mid,n} : a_r \leftrightsquigarrow a_{mid,1} \leftrightsquigarrow a_{mid,2} \leftrightsquigarrow ... \leftrightsquigarrow a_{mid,n} \leftrightsquigarrow a_l \Longrightarrow$
$\exists a_{mid,1}, a_{mid,2}, ..., a_{mid,n}, a_{u,1}, a_{u,2}, ..., a_{u,n+1} : \{a_{u,1}, a_{u,2}, ..., a_{u,n}\} \subseteq (U \cup R) \wedge (a_{u,1} \rightsquigarrow a_r \wedge a_{u,1} \rightsquigarrow a_{mid,1}) \wedge (a_{u,2} \rightsquigarrow a_{mid,1} \wedge a_{u,2} \rightsquigarrow a_{mid,2}) \wedge ... \wedge ((a_{u,n+1} \rightsquigarrow a_{mid,n} \wedge a_{u,n+1} \rightsquigarrow a_l))$.
The above statement is true in G' because $E \subseteq E'$. Since $\forall a_x, a_y : a_x \in (U \cup R) \wedge a_y \in V \wedge a_x \rightsquigarrow a_y \Longrightarrow a_y \rightsquigarrow a_x$ in G', we know $\exists a_{mid,1}, a_{mid,2}, ..., a_{mid,n}, a_{u,1}, a_{u,2}, ..., a_{u,n+1} : \{a_{u,1}, a_{u,2}, ..., a_{u,n}\} \subseteq (U \cup R) \wedge a_r \rightsquigarrow a_{u,1} \rightsquigarrow a_{mid,1} \rightsquigarrow a_{u,2} \rightsquigarrow a_{mid,2} ... \rightsquigarrow a_{mid,n} \rightsquigarrow a_{u,n+1} \rightsquigarrow a_l$.
Therefore $Live_{actor}(G,R,U) \subseteq Live_{object}(G',R')$.

Now, consider the other case that
$Live_{object}(G',R') \subseteq Live_{actor}(G,R,U)$.
For any a_r and a_l, $a_r \in R' \wedge a_l \in V' \wedge a_r \rightsquigarrow a_l$ in G',
let $\{\overline{a_{x,1}a_{u,1}}, \overline{a_{x,2}a_{u,2}}, ..., \overline{a_{x,n}a_{u,n}}\} \subseteq (E' - E)$ and be part of $a_r \rightsquigarrow a_l$ in G', such that
$(a_r \rightsquigarrow a_{x,1} \wedge \overline{a_{x,1}a_{u,1}} \in (E' - E)) \wedge (a_{u,1} \rightsquigarrow a_{x,2} \wedge \overline{a_{x,2}a_{u,2}} \in (E' - E)) \wedge ...$
$(a_{u,n-1} \rightsquigarrow a_{x,n} \wedge \overline{a_{x,n}a_{u,n}} \in (E' - E)) \wedge a_{u,n} \rightsquigarrow a_l$.
Since a reference $\overline{a_q a_p}$ in $(E' - E)$ implies that $a_p \in (U \cup R) \wedge a_q \in V \wedge a_p \rightsquigarrow a_q$, we get $a_r \rightsquigarrow a_r \wedge (a_r \rightsquigarrow a_{x,1} \wedge a_{u,1} \rightsquigarrow a_{x,1}) \wedge (a_{u,1} \rightsquigarrow a_{x,2} \wedge a_{u,2} \rightsquigarrow a_{x,2}) \wedge ...$
$\wedge (a_{u,n-1} \rightsquigarrow a_{x,n} \wedge a_{u,n} \rightsquigarrow a_{x,n}) \wedge a_{u,n} \rightsquigarrow a_l \Longrightarrow$
$a_r \leftrightsquigarrow a_{x,1} \wedge a_{x,1} \leftrightsquigarrow a_{x,2} \wedge ... \wedge a_{x,n-1} \leftrightsquigarrow a_{x,n} \wedge a_{x,n} \leftrightsquigarrow a_l \Longrightarrow$
$a_r \leftrightsquigarrow^* a_l$.
Therefore $Live_{object}(G',R') \subseteq Live_{actor}(G,R,U)$.

Transformation by Indirect Back-Pointers to Unblocked Actors The indirect back-pointer transformation method transforms actor garbage collection into passive object garbage collection by making the reference set $E' = E \cup \{\overline{a_q a_p} \mid a_u \in (U \cup R) \wedge \overline{a_p a_q} \in E \wedge a_u \rightsquigarrow a_p\}$. Figure 4 shows what the actor reference graph in Figure 1 transforms into. Notice that $\overline{a_q a_p} \in E'$ implies that $\overline{a_p a_q} \in E$. Therefore the total number of back-pointers must be no bigger than

the total number of edges in the actor reference graph G. Theorem 2 shows that the indirect back-pointer transformation method is correct. Since the correctness proof for the indirect back-pointer transformation method is very similar to the proof for the direct back-pointer transformation method, we will leave it to the readers.

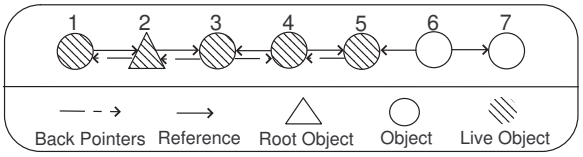

Fig. 4. An example of transformation using indirect back-pointers to unblocked actors.

Theorem 2. *Indirect back-pointer transformation.*
Let the actor reference graph be $G = \langle V, E \rangle$*,* R *be the set of roots, and* U *be the set of unblocked actors. Let* $E' = E \cup \{\overline{a_q a_p} \mid \exists a_u : a_u \in (U \cup R) \wedge \overline{a_p a_q} \in E \wedge a_u \rightsquigarrow a_p\}$*, and* $G' = \langle V, E' \rangle$ *and* $R' = R$*.*

$$Live_{actor}(G, R, U) = Live_{object}(G', R')$$

5 Implementation and Experimental Results

Theorem 1 or Theorem 2 can directly turn into an actor garbage collection algorithm by simply adding new back-pointers in the actor reference graph. Theorem 2 is a better choice to model the back-pointer algorithm because it generates fewer back-pointers.

Based on Theorem 1, we also propose the *N-color algorithm*. The idea comes from the strategy of turning "a back-pointer to an unblocked/root actor" into a special color, that is, two actors could have the same color if both of them have back-pointers to the same unblocked/root actor. A root color is defined as Color 0. To avoid multiple colors in an actor, only one color is allowed in each actor. Once different colors conflict in an actor, we combine the colors by using disjoint set operations [19]. Since any color conflict implies the two representive unblocked/root actors have the relation of $\rightsquigarrow^*$, we can conclude that actors marked by any color in a disjoint set containing a root color are live. Actors marked by the same colors may talk to each other because they are directly reachable from the same unblocked actor. Color conflict implies the may-transitively-talk-to relationship. Therefore, combining conflict colors is equivalent to grouping the set of actors that may transitively talk to each other. If a root color is included in this set, every actor in the set may transitively talk to

the root. Take Figure 5 for example. Actors marked by Colors 0, 1, or 2 are live. Unmarked actors and actors with Color 3 are garbage. Color 0 can be viewed as a back-pointer to the root; Color 1 represents a back-pointer to unblocked Actor 1; Color 2 represents a back-pointer to unblocked Actor 4; Color 3 represents a back-pointer to unblocked Actor 7.

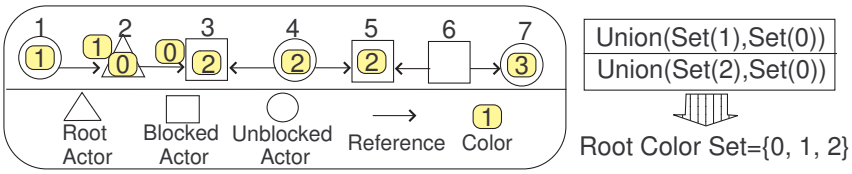

Fig. 5. An example for the N-color algorithm.

Let N be the number of actors, E the number of references in the system, and M be the number of unblocked actors. The time complexity of the back-pointer algorithm is $O(N + E)$, and its extra space complexity is $O(N + E)$. The time complexity of the algorithm is $O(N + E \lg^* M)$, and its extra space complexity is $O(M + N)$ where $O(M)$ is for the disjoint set operations and $O(N)$ is for marking.

We implement the N-color algorithm, the back-pointer algorithm, and the Vardhan-Agha transformation method on SALSA 2.0 [20]. SALSA is implemented by Java and all SALSA programs are compiled into Java source code. Figure 6 shows the total execution time of executing a SALSA Fibonacci program with different algorithms. In the experiment, the actor garbage collector is triggered every one second to do a complete scan of garbage and to reclaim all garbage and each scan may involve tens of thousands of actors and messages. The result shows that the N-color algorithm is less intrusive than the back-pointer algorithm, which is less intrusive than the Vardhan-Agha algorithm. We also measure the average execution time of marking N live actors using our algorithms and the Vardhan-Agha algorithm, as shown in Figure 7. The left side is the result of marking N actors and N references, and the right side shows the result of marking N actors and 2N references. All the results suggest that the N-Color algorithm is the best among them, and the back-pointer algorithm is the second. The result can attribute to: (1) the N-Color algorithm demands less expensive memory operations, and (2) the function lg^*M grows extremely slow, and therefore $O(N + E \lg^* M)$ is very close to $O(N + E)$ in practice.

6 Conclusions

Both passive object garbage collection and actor garbage collection can be represented as graph problems. However, the traditional root-reachability condition

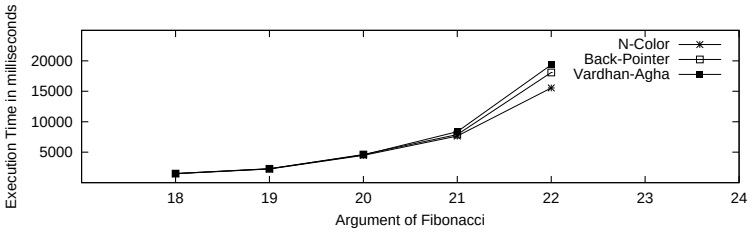

Fig. 6. Performance evaluation using a SALSA Fibonacci number program.

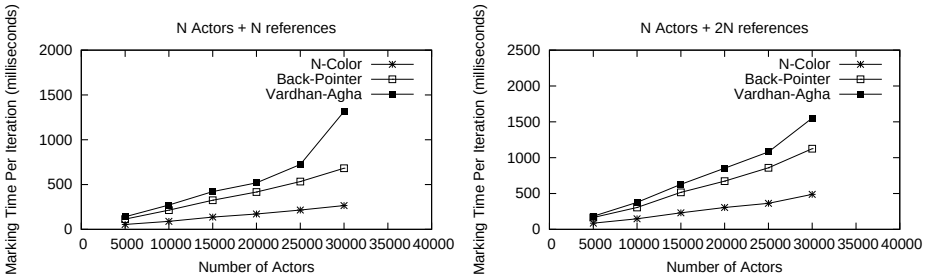

Fig. 7. Execution time of marking N actors.

that determines live objects in object graphs does not correctly detect live actors in an actor graph. Since there has been significant research in object garbage collection, developing transformation methods from actor to object graphs is benefitial for high-level actor programming language implementation.

This paper has made some contributions. First, we introduced the *may-talk-to* ($\rightsquigarrow$) and the *may-transitively-talk-to* ($\rightsquigarrow^*$) relations to explain and formally define potential communication between actors. Second, we proved the equivalence of actor garbage collection and passive object garbage collection in which we showed two vertex-preserving transformation methods to transform actor garbage collection into passive object garbage collection. Third, we presented two practical actor garbage collection algorithms, the N-color algorithm and the back-pointer algorithm. Experimental results show that they are efficient, and perform better than the Vardhan-Agha algorithm.

Acknowledgments

The authors would like to thank Yousaf Shah and Ping Wang of Rensselaer Polytechnic Institute, USA, for their comments and suggestions. This work was supported in part by the Taiwan National Science Council under Grant NSC-98-2221-E-008-080.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press (1986)
2. Hewitt, C.: Viewing control structures as patterns of passing messages. Journal of Artificial Intelligence **8**(3) (June 1977) 323–364
3. Varela, C.A., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. ACM SIGPLAN Notices. OOPSLA'2001 ACM Conference on Object-Oriented Systems, Languages and Applications **36**(12) (December 2001) 20–34
4. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the jvm platform: a comparative analysis. In: PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, New York, NY, USA, ACM (2009) 11–20
5. Moors, A., Piessens, F., Odersky, M.: Generics of a higher kind. In: OOPSLA '08, New York, NY, USA, ACM (2008) 423–438
6. ERights.org: The E Programming Language (2009) http://ERights.org/.
7. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. 2nd edn. Prentice Hall (1996)
8. Kafura, D., Washabaugh, D., Nelson, J.: Garbage collection of actors. In: OOPSLA'90 ACM Conference on Object-Oriented Systems, Languages and Applications, ACM Press (October 1990) 126–134
9. Vardhan, A., Agha, G.: Using passive object garbage collection algorithms for garbage collection of active objects. In: ISMM'02. ACM SIGPLAN Notices, Berlin, ACM Press (June 2002) 106–113
10. Wang, W., Varela, C.A.: Distributed garbage collection for mobile actor systems: The pseudo root approach. Lecture Notes in Computer Science **3947** (May 2006) 360–372
11. Jones, R.E.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, Chichester (July 1996) With a chapter on Distributed Garbage Collection by R. Lins.
12. Abdullahi, S.E., Ringwood, A.: Garbage collecting the internet: A survey of distributed garbage collection. ACM Computing Surveys **30**(3) (1998) 330–373
13. Halstead, R.H.: Reference Tree Networks: Virtual Machine and Implementation. PhD thesis, MIT Laboratory for Computer Science (July 1979) Technical report MIT/LCS/TR–222.
14. Baker, H.G.: List processing in real-time on a serial computer. Communications of the ACM **21**(4) (1978) 280–294
15. Lieberman, H., Hewitt, C.: A real-time garbage collector based on the lifetimes of objects. Commun. ACM **26**(6) (1983) 419–429
16. Washabaugh, D.: Real-time garbage collection of actors in a distributed system. Master's thesis, Virginia Tech, Blacksburg, VA (February 1990)
17. Nelson, J.: Automatic, incremental, on-the-fly garbage collection of actors. Master's thesis, Virginia Tech, Blacksburg, VA (February 1989)
18. Dickman, P.: Incremental, distributed orphan detection and actor garbage collection using graph partitioning and Euler cycles. Lecture Notes in Computer Science **1151** (October 1996) 141–158
19. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: 21. In: Introduction to Algorithms. Second edn. MIT Press/McGraw-Hill (2001) 498–522
20. Worldwide Computing Laboratory: The SALSA Programming Language (2009) http://wcl.cs.rpi.edu/salsa/.