A Non-Blocking Reference Listing Algorithm for Mobile Active Object Garbage Collection

Wei-Jen Wang and Carlos A. Varela
Department of Computer Science, RPI

Automatic garbage collection (GC) gives abstraction to distributed application development, promoting code quality and improving resource management. Unreachability of *active objects* or *actors* from the root set is not a sufficient condition to collect actor garbage, making passive object GC algorithms unsafe when directly used on actor systems. In practical actor languages, all actors have references to the root set since they can interact with users, *e.g.*, through standard input or output streams. This feature makes every unblocked actor live, and thus we call it *the live unblocked actor principle*. Following this idea, we introduce *pseudo-roots*: a dynamic set of actors that can be viewed as the root set. Pseudo-roots use protected (undeletable) references to ensure that no actors are erroneously collected even with messages in transit. The pseudo-root approach simplifies distributed actor garbage collection by representing in-transit messages in the actor reference graph even if both application and system messages are unordered (non-FIFO) and communication is asynchronous. Our algorithm also supports systems not following the live unblocked actor principle and therefore, is applicable to more traditional actor garbage collection. We formalize the computing model of the pseudo-root approach and provide proofs of correctness. Furthermore, we summarize empirical results on the performance and scalability of distributed GC using a particle physics maximum likelihood evaluation fitter on a 72-processor cluster environment.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Memory management*; D.4.2 [**Operating Systems**]: Storage Management—*Garbage collection*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Distributed garbage collection, Reference listing, Actors, Active objects, Mobile objects

1. INTRODUCTION

Large applications running on the grid, or on the internet, require runtime reconfigurability for better performance, *e.g.*, relocating application sub-components to improve locality without affecting the semantics of the distributed system. A runtime reconfigurable distributed system can be easily defined by the actor model of computation [Agha 1986; Hewitt, C. 1977]. The actor model provides a unit of encapsulation for a thread of control along with internal state (see Figure 1). An actor is either *unblocked* or *blocked*. It is unblocked if it is processing a message or has messages in its message box, and it is blocked otherwise. Communication between actors is purely asynchronous: non-blocking and non-First-In-First-Out (non-FIFO). However, communication is guaranteed: all messages are eventually and fairly delivered. In response to an incoming message, an actor can use its

Author's address: W. Wang and C. Varela, Dept. of Computer Science, RPI, 110 8th street, Troy, NY 12180.

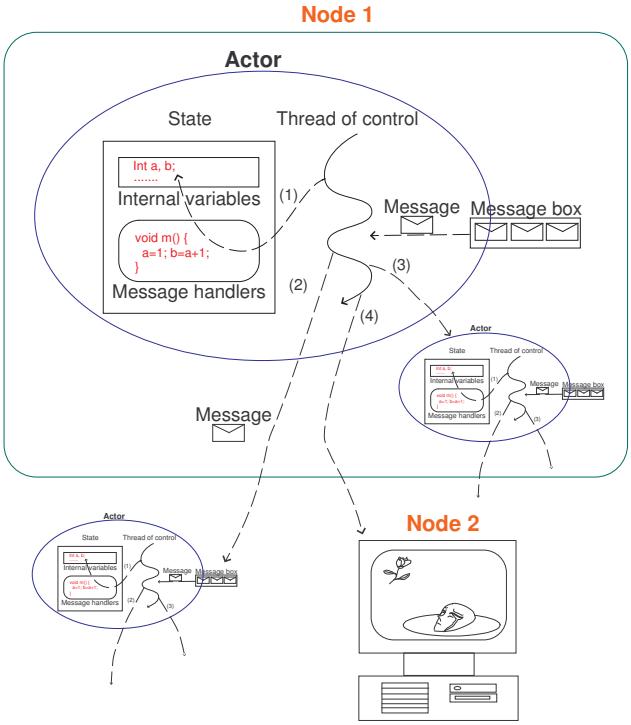

Fig. 1. An actor is a reactive entity which communicates with others by asynchronous non-FIFO messages. In response to an incoming message, it can use its thread of control to 1) modify its internal state, 2) send messages to other actors, 3) create actors, or 4) migrate to another host.

thread of control to modify its encapsulated internal state, send messages to other actors, create actors, or migrate to another host.

Many programming languages have partial or total support for actor semantics, such as SALSA [Varela and Agha 2001], ABCL [Yonezawa 1990], THAL [Kim 1997], and Erlang [Armstrong et al. 1996]. Some libraries also support actor creation and use in object-oriented programming languages, such as the Actor Foundry [Open Systems Lab 1998] and ProActive [Baduel et al. 2006] for Java, Actalk [Briot 1989] for Smalltalk, and mobile MPI processes [El Maghraoui et al. 2005] for C++. In designing these languages or systems, memory reuse becomes an important issue to support dynamic data structures — such as linked lists. Automatic garbage collection is the key to enable memory reuse and to reduce programmers' efforts on their error-prone manual memory management.

The problem of distributed garbage collection (GC) is difficult because of: 1) information distribution, 2) lack of a global clock, 3) concurrent activities, and 4) possible failures of the network or computing nodes. These factors complicate detection of a consistent global state of a distributed system. Comparing to object-oriented systems, a pure actor system demands automatic GC as well, even more,

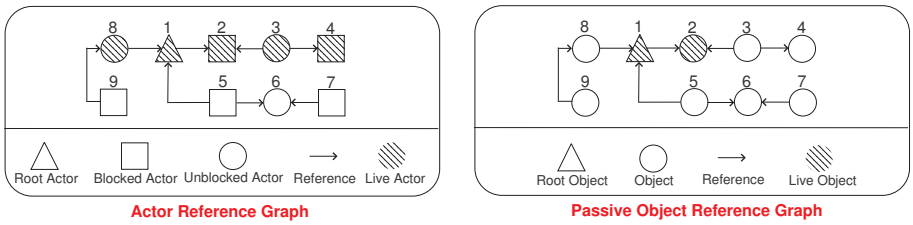

Fig. 2. Actors 3, 4, and 8 are live because they can potentially send messages to the root. Objects 3, 4, and 8 are garbage because they are not reachable from the root.

because of its distributed, mobile, and resource-consuming nature. Actor GC is traditionally considered as a harder problem than passive object GC because of two additional difficulties to overcome:

(1) Simply following the references from the root set of actors does not work in the actor GC model. Figure 2 explains an important difference between actor GC and passive object GC: an actor's internal thread or control may render it live even if unreachable from the root set.

(2) Unordered asynchronous message delivery complicates actor GC. Most existing algorithms cannot tolerate out-of-order messages.

Previous distributed GC algorithms (including actor GC algorithms) rely on First-In-First-Out (FIFO) communication which simplifies detection of a consistent global state. A distributed object GC algorithm either adopts: 1) a lightweight reference counting/listing approach which cannot collect distributed mutually referenced data structures (*cycles*), 2) a trace-based approach which requires a consistent state of a distributed system, or 3) a hybrid approach [Abdullahi and Ringwood 1998].

Actor mobility is another new challenge to overcome. The concept of *in-transit* actors complicates the design of actor communication — locality of actors can change, which means even simulated FIFO communication with message redelivery is impractical, or at least limits concurrency by unnecessarily waiting for message redelivery. FIFO communication is an assumption of existing distributed GC algorithms. For instance, distributed reference counting algorithms demand FIFO communication to ensure that a reference-deletion system message does not precede any application messages.

In actor-oriented programming languages, an actor must be able to access resources which are encapsulated in service actors. To access a resource, an actor requires a reference to it. This implies that actors keep persistent references to some special service actors — such as the file system service and the standard output service. Furthermore, an actor can explicitly create references to public services. For instance, an actor can dynamically convert a string into a reference to communicate with a service actor, analogous to accessing a web service by a web browser using a URL.

This paper extends [Wang and Varela 2006a] by considering actor garbage collection without assuming the live unblocked actor principle holds. Furthermore, we provode a computing mode and its correctness proofs. The algorithm differs from previous actor GC models by introducing: 1) asynchronous, unordered message delivery of both application messages and system messages, 2) resource access rights, and 3) actor mobility.

Outline of This Paper

The remainder of the paper is organized as follows: In Section 2 we give the definition of garbage actors. In Section 3 we describe the pseudo-root approach — a reference listing algorithm for mobile actor garbage collection. In Section 4 we present our implementation of the pseudo-root approach. In Section 5 we introduce the formal computing model of the pseudo-root approach, along with its correctness proofs. In Section 6 we show the experimental results. In Section 7 we discuss related work. Section 8 contains concluding remarks and future work.

2. GARBAGE IN ACTOR SYSTEMS

The definition of actor garbage comes from the idea of whether an actor is doing meaningful computation. Meaningful computation is defined as having the ability to communicate with any of the *root actors*, that is, to access any resource or public service. The widely used definition of live actors is described in [Kafura et al. 1990]. Conceptually, an actor is live if it is a root or it can either potentially: 1) receive messages from the root actors or 2) send messages to the root actors. The set of actor garbage is then defined as the complement of the set of live actors. To formally describe our new actor GC model, we introduce the following definitions:

—**Blocked actor**: An actor is blocked if it has no pending messages in its message box, nor any message being processed. Otherwise it is *unblocked*.

—**Reference**: A reference indicates an address of an actor. Actor a_p can only send messages to Actor a_q if a_p has a reference pointing to a_q, denoted as $\overline{a_p a_q}$.

—**Inverse reference**: An inverse reference is a conceptual reference in the counter-direction of an existing reference.

—**Acquaintance**: Let Actor a_p have a reference pointing to Actor a_q. a_q is an acquaintance of a_p, and a_p is an *inverse acquaintance* of a_q.

—**Root actor**: An actor is a root actor if it encapsulates a resource, or if it is a public service — such as I/O devices, web services, and databases.

The original definition of live actors is denotational because it uses the concept of "potential" message delivery and reception. To make it more operational, we assume instant message delivery and use the term "*potentially live*" [Dickman 1996] to define live actors.

—**Potentially live actors**:
  —Every unblocked actor and root actor is potentially live.
  —Every acquaintance of a potentially live actor is potentially live.

—**Live actors**:
  —A root actor is live.

—Every acquaintance of a live actor is live.
—Every potentially live, inverse acquaintance of a live actor is live.

3. THE PSEUDO-ROOT APPROACH

Together with *reference listing*, the core concept of the pseudo-root approach is to integrate message delivery and reference passing into the reference graph representation — *sender pseudo-roots* and *protected references*. It thus enables the use of unordered, asynchronous, and non-instantaneous communication.

The pseudo-root approach was mainly devised for actor programming languages, which abide by the *live unblocked actor principle* — a principle which says every unblocked actor should be treated as a live actor. Nevertheless, the pseudo-root approach can also be used for actor marking algorithms without the assumption of the live unblocked actor principle.

The Live Unblocked Actor Principle

Without program analysis techniques, the ability of an actor to access resources provided by an actor-oriented programming language implies explicit reference creation to access service actors. The ability to access local service actors (e.g. the standard output) and explicit reference creation to public service actors make the following statement true: "*every actor has persistent references to root actors*". This statement is important because it changes the meaning of actor GC, making actor GC similar to passive object GC. It leads to the *live unblocked actor principle*, which says every unblocked actor is live. The live unblocked actor principle is easy to prove. Since each unblocked actor is: 1) an inverse acquaintance of the root actors and 2) defined as potentially live, it is live according to the definition of actor GC.

With the live unblocked actor principle, every unblocked actor can be viewed as a root. Liveness of blocked actors depends on the transitive reachability from unblocked actors and root actors. Without considering in-transit messages, a blocked actor, which is transitively reachable from an unblocked actor or a root actor, is defined as potentially live. With persistent root references, such potentially live, blocked actors are live because they are inverse acquaintances of some root actors. This idea leads to the core concept of *pseudo-root* actor GC.

Pseudo-Root Actor Garbage Collection

It is impossible to ignore in-transit messages in a distributed system and correctly perform actor garbage collection simply based on the actor reference graph. As a consequence, we introduce the concept of sender pseudo-root actors and protected references for actor garbage collection. Pseudo-root actor GC starts by identifying some live (not necessarily root) or even garbage actors as pseudo-roots. There are three kinds of pseudo-root actors: 1) root actors, 2) unblocked actors, and 3) *sender pseudo-root actors*. The sender pseudo-root actor refers to an actor which has sent a message and the message has not yet been received. The goal of sender pseudo-roots is to prevent erroneous garbage collection of actors, either targets of in-transit messages or whose references are part of in-transit messages. A sender pseudo-root always contains at least one protected reference — a reference that has been used to deliver messages which are currently in transit, or a reference to

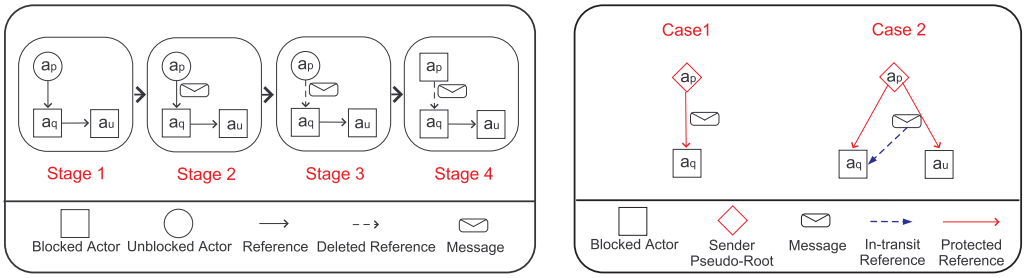

Fig. 3. The left side of the figure shows a possible race condition of mutation and message passing. The right side of the figure illustrates both kinds of sender pseudo-root actors.

represent an actor referenced by an in-transit message — which we call an *in-transit reference*. A protected reference cannot be deleted until the message sender knows the in-transit messages have been received correctly.

Asynchronous communication introduces the following problem (see the left side of Figure 3): application messages from Actor a_p to Actor a_q can be in transit, but the reference held by Actor a_p can be removed. Stage 3 shows that Actor a_q and a_u are likely to be erroneously reclaimed, while Stage 4 shows that all of the actors are possibly erroneously reclaimed. Our solution is to temporarily keep the reference to Actor a_q undeleted and identify Actor a_p as live (Case 1 of the right side of Figure 3). This approach guarantees liveness of Actor a_q by tracing from Actor a_p. Actor a_p is named the *sender pseudo-root* because it has an in-transit message to Actor a_q and it is not a real root. Furthermore, it can be garbage but cannot be collected. The reference from a_p to a_q is protected and a_p is considered live until a_p knows that the in-transit message is delivered.

To prevent erroneous GC, actors pointed by in-transit references must unconditionally remain live until the receiver receives the message. A similar solution can be re-used to guarantee the liveness of the referenced actor: the sender becomes a sender pseudo-root and keeps the reference to the referenced actor undeleted (Case 2).

Using pseudo-roots, *the persistent references to roots can be ignored*. Figure 4 illustrates an example of the mapping of pseudo-root actor GC. We can now safely ignore: 1) dynamic creation of references to public services and 2) persistent references to local services.

Imprecise Inverse Reference Listing

In a distributed environment, an inter-node referenced actor must be considered live from the perspective of local GC because it can possibly receive a message from a remote actor. To know whether an actor is inter-node referenced, each actor should maintain inverse references to indicate if it is inter-node referenced. This approach is usually called *reference listing*. Maintaining precise inverse references in an asynchronous way is performance-expensive. Fortunately, imprecise inverse references are acceptable if all inter-node referenced actors can be identified as live — an inter-node referenced actor can be a new kind of pseudo-root actor (*the*

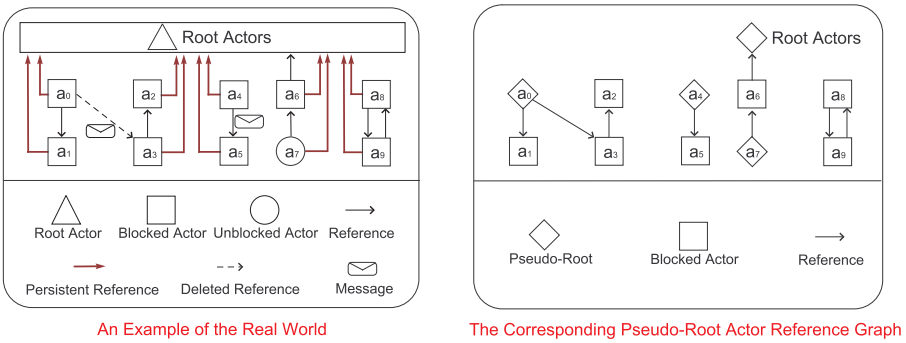

Fig. 4. An example of pseudo-root actor garbage collection which maps the real state of the given system to a pseudo-root actor reference graph.

global pseudo-root), or transitively reachable from some local pseudo-root actors to guarantee their liveness.

Actor Garbage Collection without the Live Unblocked Actor Principle

Without the live unblocked actor principle, the pseudo-root approach still supports actor garbage collection in a distributed environment by changing two definitions as follows:

—Sender pseudo-roots should be considered as unblocked actors.

—Global pseudo-roots should be considered as roots, and thus we call them *global roots*. There are two kinds of global roots. The first kind of global root actor, namely *the unblocked-reachable global root*, is transitively reachable from an unblocked actor and has a reference pointing to a remote (or migrating) actor. It must be considered live because it can possibly send a message to a remote root actor. The second type of actor, *the remotely referenced global root*, has an inverse reference pointing to a remote actor. The pseudo-root approach cannot directly identify the unblocked-reachable global roots, which must be handled by an actor marking algorithm.

The pseudo-root approach in actor garbage collection also guarantees that if an actor is remotely referenced, it either has an inverse reference to figure out it is remotely referenced (the remotely referenced global root), or it is reachable from an unblocked actor and the unblocked actor can reach an unblocked-reachable global root. Once an actor marking algorithm follows the new definitions to identify live actors, all local garbage can be reclaimed, including the actors which are potentially live and yet garbage.

4. IMPLEMENTATION OF THE PSEUDO-ROOT APPROACH

To implement the pseudo-root approach, we propose the *actor garbage detection protocol*. The actor garbage detection protocol, implemented as part of the SALSA programming language [Varela and Agha 2001; Worldwide Computing Laboratory

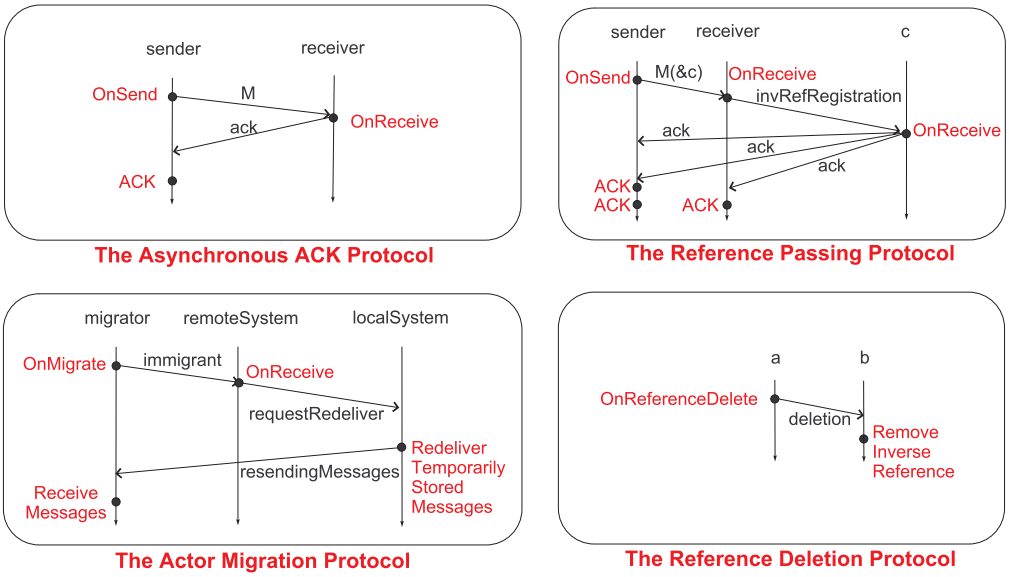

Fig. 5. The actor garbage detection sub-protocols.

2007], consists of four sub-protocols — the *asynchronous ACK protocol*, the *reference passing protocol*, the *migration protocol*, and the *reference deletion protocol*. Messages are divided into two categories — the *application messages* which require asynchronous acknowledgements, and the *system messages* that not always require an acknowledgement.

The Asynchronous ACK Protocol

The asynchronous ACK protocol is designed to help identifying sender pseudo-roots. Each reference maintains a counter, the *expected acknowledgement count*. A reference can be deleted only if its expected acknowledgement count is zero. An actor is a sender pseudo-root if the total expected acknowledgements of its references is greater than zero. The protocol is shown in the left upper part of Figure 5, in which actor `sender` sends a message to actor `receiver`. The event handler `OnSend` is triggered when an application message is sent; the event handler `OnReceive` is invoked when a message is received. If a message to receive requires an acknowledgement, the event handler `OnReceive` will generate an acknowledgement to the message sender. The asynchronous message handler `ACK` is concurrently executed by an actor to decrease the expected acknowledgement count of the reference to actor `receiver` held by actor `sender`. With the asynchronous ACK protocol, the garbage collector can identify sender pseudo-roots and protected references from the perspective of implementation:

—A *protected reference* is one whose expected acknowledgement count is greater than zero. A protected reference cannot be deleted.

—A *sender pseudo-root* is one which has at least one protected reference.

The Reference Passing Protocol

The reference passing protocol specifies how to build inverse references in an asynchronous manner. A typical scenario of reference passing is to send a message `M` containing a reference to `c`, from `sender` to `receiver`. Reference *sender receiver* and Reference *sender c* are protected at the beginning by increasing their expected acknowledgement counts. Then `sender` sends the application message `M` to `receiver`. Right after `receiver` has received the message, it generates a special system message `invRefRegistration` to `c` to register the corresponding inverse reference of Reference *receiver c* in `c`. Requiring `invRefRegistration` to be acknowledged is to ensure that reference deletion of reference `(receiver,c)` always happens after `c` has built the corresponding inverse reference. Two special acknowledgements from `c` to `sender`, which can be combined into one, are then sent to decrease the counts of the protected references *sender c* and *sender receiver*. The protocol is shown in the right upper side of Figure 5.

The Migration Protocol

Implementation of the migration protocol requires assistance from two special actors, `remoteSystem` at a remote computing node, and `localSystem` at the local computing node. An actor migrates by encoding itself into a message, and then delivers the message to `remoteSystem`. During this period, messages to the migrating actor are stored at `localSystem`. After migration, `localSystem` delivers the temporarily stored messages to the migrated actor asynchronously. Every migrating actor becomes a pseudo-root by increasing the expected acknowledgement count of its self reference. The migrating actor decreases the expected acknowledgement count of its self reference when it receives the temporarily stored messages. The protocol can be simply viewed as that the migrating actor sends a message to itself, and it will receive the message until it has finished migration. The protocol is shown in the left lower side of Figure 5.

The Reference Deletion Protocol

A reference can be deleted if it is not protected — its expected acknowledgement count must be zero. The deletion automatically creates a system message to the acquaintance of the actor deleting the reference to remove the inverse reference held by the acquaintance. The protocol is shown in the right lower side of Figure 5.

5. COMPUTING MODEL AND THE PSEUDO-ROOT APPROACH PROPERTIES

In this section, we define the model of the pseudo-root approach, and then prove the safety and liveness properties of the pseudo-root approach.

5.1 Computing Model

To implement the concept of protected references, we introduce the data structure of actor references, which consists of three elements — the source and target addresses to describe which actor holds a reference to another actor, a counter to track the expected incoming acknowledgements, and a boolean variable to indicate if a reference has been deleted at the application level. Then we define a set of messages along with their explanations. These notations help define the actor system with

the pseudo-root approach, which is an abstract machine with a given initial state, identified by a set of actor names, a map from actor references to their meta-data, a set of inverse references, and a set of in-transit messages.

DEFINITION 5.1. *The meta-data map of actor references.*
A meta-data map (function) of actor references is described as

$$R \equiv \{\overline{a_0 a_1} \mapsto \langle n_0, d_0\rangle, \overline{a_2 a_3} \mapsto \langle n_1, d_1\rangle, ...\}.$$

Its domain, $dom(R) = \{\overline{a_0 a_1}, \overline{a_2 a_3}, ...\}$*, is a set of actor references while its range,* $\{\langle n, d\rangle \mid n \in \mathbb{N} \wedge d \in \{true, false\}\}$*, is the meta-data of the actor reference, where*

—*n is an integer to keep track of expected incoming acknowledgements, and*
—$d \in \{true, false\}$*, where* $false$ *denotes a normal reference, and* $true$ *denotes a deleted reference at the application level.*

DEFINITION 5.2. *The extended meta-data map of actor references.*
For a given meta-data map of references R*,* $R' = R[\overline{a_i a_j} \mapsto \langle n, d\rangle]$ *is defined as the extended amp, such that* $R'(\overline{a_i a_j}) = \langle n, d\rangle$ *and* $R'(x) = R(x)$ *for* $x \neq \overline{a_i a_j}$*.*

DEFINITION 5.3. *Message*
Let x be a unique id. A message

msg	$\equiv$	$m_x\langle a_i, a_j\rangle$	*(Regular message from* a_i *to* a_j*.)*
	|	$mack_x\langle a_i, a_j\rangle$	*(Acknowledgement for* $\overline{a_i a_j}$*.)*
	|	$mr_x\langle a_h, a_i, a_j\rangle$	*(Reference passing, from* a_h *to* a_i *with a reference to* a_j*.)*
	|	$mir_x\langle a_h, a_i, a_j\rangle$	*(Inverse reference registration, initialized by* a_h *to register* $\overline{a_i a_j}$*.)*
	|	$md_x\langle a_i, a_j\rangle$	*(Deletion message of the inverse reference of* $\overline{a_i a_j}$*.)*
	|	$mm_x\langle a_i\rangle$	*(Migration of* a_i*.)*

DEFINITION 5.4. *Actor system configuration*
An actor system configuration

$$S \equiv \langle A, R, IR, M\rangle,$$

is a 4-tuple where

—*A is the set of actor names,* $\{a_0, a_1, ...\}$*.*
—*R is the map from actor references to their meta-data.*
—*IR is the set of inverse references,* $\{\overline{a_0 a_1}, \overline{a_2 a_3}, ...\}$*.*
—*M is the set of messages,* $\{msg_0, msg_1, ...\}$*.*

The initial state of S is defined as

$$\langle\{a_{init}\}, [\overline{a_{init} a_{init}} \mapsto \langle 0, false\rangle], \{\overline{a_{init} a_{init}}\}, \emptyset\rangle.$$

A state of the actor system can evolve to another state according to a set of rules, usually called a labeled transition system (LTS). Following the actor model, the transition rules of the proposed abstract machine can be classfied into two categories. The first type consists of *the spontaneous transitions* by unblocked actors, including actor creation, reference passing, actor-exit (to migrate out), reference

deletion (application or GC level), and message sending. The other type of transitions, *the message-driven transitions*, is the consequent transition triggered by the spontaneous transition, including actor-enter (to migrate into), reference reception, inverse reference registration, acknowledgement, and inverse reference deletion. We define two different reference deletion transition rules because of protected references — a deleted protected reference must remain visible to garbage collectors but invisible to applications — The application-level reference deletion transition makes the reference invisible to the application, while the GC-level reference deletion transition deletes the reference physically and will trigger the deletion of the corresponding inverse reference. The label of a transition rule follows the format, $op(p_1, p_2, ...)$, where op is the name of the transition rule and $p_1, p_2, ...$ are parameters. The transition rules are shown as follows:

DEFINITION 5.5. *Transitive relationship on actor system configurations. The transition notation, $\rightarrow$, is defined as follows:*

—*Spontaneous transitions:*

—$AC(a_i, a_j)$*:* Actor creation of a_j by a_i.
$\langle A, R, IR, M\rangle \xrightarrow{AC(a_i,a_j)}$
$\langle A \cup \{a_j\}, R[\overline{a_i a_j} \mapsto \langle 0, false\rangle, \overline{a_j a_j} \mapsto \langle 0, false\rangle], IR \cup \{\overline{a_i a_j}, \overline{a_j a_j}\}, M\rangle$,
where a_j is fresh $\wedge$ $a_i \in A$.

—$RC1(a_h, a_i, a_j)$*:* Reference passing from a_h to a_i with a reference to a_j.
$\langle A, R[\overline{a_h a_i} \mapsto \langle n_0, false\rangle, \overline{a_h a_j} \mapsto \langle n_1, false\rangle], IR, M\rangle \xrightarrow{RC1(a_h,a_i,a_j)}$
$\langle A, R[\overline{a_h a_i} \mapsto \langle n_0 + 1, false\rangle, \overline{a_h a_j} \mapsto \langle n_1 + 1, false\rangle], IR, M \cup \{mr_x\langle a_h, a_i, a_j\rangle\}\rangle$,
where x is fresh $\wedge$ $\{a_h, a_i, a_j\} \subseteq A$.

—$MM1(a_i)$*:* Actor-exit of a_i.
$\langle A, R[\overline{a_i a_i} \mapsto \langle n, false\rangle], IR, M\rangle \xrightarrow{MM1(a_i)}$
$\langle A - \{a_i\}, R[\overline{a_i a_i} \mapsto \langle n + 1, false\rangle], IR, M \cup \{mm_{x1}\langle a_i\rangle, mack_{x2}\langle a_i, a_i\rangle\}\rangle$,
where $x1$ and $x2$ are fresh $\wedge$ $a_i \in A$.

—$MS1(a_i, a_j)$*:* Message sending from a_i to a_j.
$\langle A, R[\overline{a_i a_j} \mapsto \langle n, false\rangle], IR, M\rangle \xrightarrow{MS1(a_i,a_j)}$
$\langle A, R[\overline{a_i a_j} \mapsto \langle n + 1, false\rangle], IR, M \cup \{m_x\langle a_i, a_j\rangle\}\rangle$,
where x is fresh $\wedge$ $\{a_i, a_j\} \subseteq A$.

—$RDA(a_i, a_j)$*:* Application-level reference deletion of a_j at a_i.
$\langle A, R[\overline{a_i a_j} \mapsto \langle n, false\rangle], IR, M\rangle \xrightarrow{RDA(a_i,a_j)}$
$\langle A, R[\overline{a_i a_j} \mapsto \langle n, true\rangle], IR, M\rangle$,
where $\{a_i, a_j\} \subseteq A \wedge a_i \neq a_j$. [1]

—$RD1(a_i, a_j)$*:* GC-level reference deletion of a_j at a_i.
$\langle A, R[\overline{a_i a_j} \mapsto \langle 0, true\rangle], IR, M\rangle \xrightarrow{RD1(a_i,a_j)}$

[1] We assume an actor always keeps a reference to itself and thus self-reference deletion is impossible.

$\langle A, R, IR, M \cup \{md_x\langle a_i, a_j\rangle\}\rangle$,
where x *is fresh* $\wedge\ \{a_i, a_j\} \subseteq A$. [2]

—*Message-driven transitions:*

—$RC2(a_h, a_i, a_j)$*:* Reference reception of a_j by a_i, initialized by a_h.
$\langle A, R, IR, M \cup \{mr_x\langle a_h, a_i, a_j\rangle\}\rangle \xrightarrow{RC2(a_h,a_i,a_j)}$
$\langle A, R[\overline{a_i a_j} \mapsto \langle 1, false\rangle], IR, M \cup \{mir_{x1}\langle a_h, a_i, a_j\rangle\}\rangle$,
where $\overline{a_i a_j}$ *and* $x1$ *are fresh* $\wedge\ \{a_h, a_i, a_j\} \subseteq A$.

—$RC3(a_h, a_i, a_j)$*:* Inverse reference registration at a_j from a_i, initialized by a_h.
$\langle A, R, IR, M \cup \{mir_x\langle a_h, a_i, a_j\rangle\}\rangle \xrightarrow{RC3(a_h,a_i,a_j)}$
$\langle A, R, IR \cup \{\overline{a_i a_j}\}, M \cup \{mack_{x1}\langle a_i, a_j\rangle, mack_{x2}\langle a_h, a_i\rangle, mack_{x3}\langle a_h, a_j\rangle\}\rangle$,
where $x1$*,* $x2$*, and* $x3$ *are fresh* $\wedge\ \{a_h, a_i, a_j\} \subseteq A$.

—$MM2(a_i)$*:* Actor-enter of a_i.
$\langle A, R, IR, M \cup \{mm_{x1}\langle a_i\rangle\}\rangle \xrightarrow{MM2(a_i)}$
$\langle A \cup \{a_i\}, R, IR, M\rangle$.

—$MS2(a_i, a_j)$*:* Message reception of a_j from a_i.
$\langle A, R, IR, M \cup \{m_x\langle a_i, a_j\rangle\}\rangle \xrightarrow{MS2(a_i,a_j)}$
$\langle A, R, IR, M \cup \{mack_{x1}\langle a_i, a_j\rangle\}\rangle$,
where $x1$ *is fresh* $\wedge\ \{a_i, a_j\} \subseteq A$.

—$RD2(a_i, a_j)$*:* Inverse reference deletion of $\overline{a_i a_j}$.
$\langle A, R, IR \cup \{\overline{a_i a_j}\}, M \cup \{md_x\langle a_i, a_j\rangle\}\rangle \xrightarrow{RD2(a_i,a_j)}$
$\langle A, R, IR, M\rangle$,
where $\{a_i, a_j\} \subseteq A$.

—$ACK(a_i, a_j)$*:* Decreasing expected acknowledgements for $\overline{a_i a_j}$.
$\langle A, R[\overline{a_i a_j} \mapsto \langle n, d\rangle], IR, M \cup \{mack_x\langle a_i, a_j\rangle\}\rangle \xrightarrow{ACK(a_i,a_j)}$
$\langle A, R[\overline{a_i a_j} \mapsto \langle n-1, d\rangle], IR, M\rangle$,
where $d \in \{true, false\} \wedge \{a_i, a_j\} \subseteq A$.

The pseudo-root actors play important roles in the pseudo-root approach since they are the starting point to identify live actors. Unblocked actors and root actors can be identified easily by garbage collectors, while the sender pseudo-root actors and the global pseudo-root actors are implementation dependent. As a consequence, definitions for the sender pseudo-root actors and the global pseudo-root actors are mandatory. We first define sender pseudo-roots, and we defer the definition of global pseudo-root actors (Definition 5.17) until we define the partial view of an actor system (Definition 5.16).

DEFINITION 5.6. *Sender pseudo-root actors.*
An actor, a_p*, is a sender pseudo-root actor in an actor system* $\langle A, R, IR, M\rangle$ *if and only if*

$$\exists a_q \in A : (R(\overline{a_p a_q}) = \langle n, d\rangle \wedge n > 0).$$

[2]GC-level reference deletion only happens when the reference to delete is removed by the application and no longer protected.

5.2 The Pseudo-Root Approach Properties

The discrete timeline of an actor system can be identified by the order of executed transitions (events) because only transitions can change the state of the actor system. The timeline helps chronological reasoning about an actor system. It is defined by a set of ordered time points:

DEFINITION 5.7. *Timeline of an actor system.*
A timeline of a given actor system consists of time points, where a time point, t, *of the timeline of the given actor system S, is the time period from the time that the t_{th} transition occurs and right before the $(t+1)_{th}$ transition happens. The original point of the timeline is* 0, *representing the time period when the initial state S_0 exists.*

The computing model of the pseudo-root approach maintains *paired references* in most cases, which means an actor reference usually has a corresponding inverse reference. Under some circumstances, some references exist without their corresponding inverse references or some inverse references exist without their corresponding actor references. However, the pseudo-root approach guarantees the following property all the time as an invariant — *if an actor a_q is pointed by an unpaired actor reference, then there exists a sender pseudo-root actor a_p which has a paired reference pointing to a_q.* To formally describe the property, we introduce the following definitions:

DEFINITION 5.8. *Paired references, unpaired references, and unpaired inverse references of an actor.*
Let $S = \langle A, R, IR, M\rangle$ be the state of the actor system at time point t. Let $a_q \in A$.

—*Paired references to a_q at time t:*
$PRS(S, a_q) \equiv \{\overline{a_p a_q} \mid \overline{a_p a_q} \in dom(R) \wedge \overline{a_p a_q} \in IR\}$.
—*Unpaired references to a_q at time t:*
$URS(S, a_q) \equiv \{\overline{a_p a_q} \mid \overline{a_p a_q} \in dom(R) \wedge \overline{a_p a_q} \notin IR\}$.
—*Unpaired inverse references from a_q at time t:*
$UIRS(S, a_q) \equiv \{\overline{a_p a_q} \mid \overline{a_p a_q} \notin dom(R) \wedge \overline{a_p a_q} \in IR\}$.

DEFINITION 5.9. *Inverse reference registration messages and inverse reference deletion messages of an actor.*
Let $S = \langle A, R, IR, M\rangle$ be the state of the actor system at time point t. Let $a_q \in A$.

—*Inverse reference registration messages for a_q at time t:*
$MIRS(S, a_q) \equiv \{mir_x\langle a_{p1}, a_{p2}, a_q\rangle \mid mir_x\langle a_{p1}, a_{p2}, a_q\rangle \in M\}$.
—*Inverse reference deletion messages for a_q at time t:*
$MDS(S, a_q) \equiv \{md_x\langle a_p, a_q\rangle \mid md_x\langle a_p, a_q\rangle \in M\}$.

There are some invariants of the pseudo-root approach which can be used for theorem proofs. Lemma 5.10 says that the total number of unpaired inverse references from a_q at S is always equal to the total number of inverse reference deletion messages for a_q at S (i.e., $|UIRS(S, a_q)| = |MDS(S, a_q)|$), which will be used in the liveness property proof.

LEMMA 5.10. *Let $S = \langle A, R, IR, M\rangle$ be an actor system configuration and $a_q \in A$. In the actor system, the following equation is true:*

$$|UIRS(S, a_q)| = |MDS(S, a_q)|.$$

PROOF. The only two transitions that can affect $|UIRS(S, a_q)|$ or $|MDS(S, a_q)|$ are $RD1$ and $RD2$. $RD1$ increments $|UIRS(S, a_q)|$ and $|MDS(S, a_q)|$ by one and $RD2$ decrements $|UIRS(S, a_q)|$ and $|MDS(S, a_q)|$ by one. Thus they don't affect the equation. Because the initial state of the actor system follows the equation and no transition can change the equation, $|UIRS(S, a_q)| = |MDS(S, a_q)|$ must be true. □

Lemma 5.11 says that the expected acknowledgement count of an actor reference can never be decremented to a negative number. In fact, it is equal to the size of a set of messages. The equation can be used to prove Lemma 5.12, which says that a reference must exist if some messages exist.

LEMMA 5.11. *The expected acknowledgement count equation.*
Let the configuration of the actor system be $S = \langle A, R, IR, M\rangle$. *Let* $R(\overline{a_i a_j}) = \langle n, d\rangle$. *Then*

$$\begin{aligned} n \quad = \quad & |\{mr_x\langle a_i, a_m, a_j\rangle \mid mr_x\langle a_i, a_m, a_j\rangle \in M\}| \quad + \\ & |\{mr_x\langle a_i, a_j, a_m\rangle \mid mr_x\langle a_i, a_j, a_m\rangle \in M\}| \quad + \\ & |\{mir_x\langle a_i, a_m, a_j\rangle \mid mir_x\langle a_i, a_m, a_j\rangle \in M\}| \quad + \\ & |\{mir_x\langle a_i, a_j, a_m\rangle \mid mir_x\langle a_i, a_j, a_m\rangle \in M\}| \quad + \\ & |\{mir_x\langle a_m, a_i, a_j\rangle \mid mir_x\langle a_m, a_i, a_j\rangle \in M\}| \quad + \\ & |\{m_x\langle a_i, a_j\rangle \mid m_x\langle a_i, a_j\rangle \in M\}| \quad + \\ & |\{mack_x\langle a_i, a_j\rangle \mid mack_x\langle a_i, a_j\rangle \in M\}| \end{aligned}$$

PROOF. The proof can be achieved by examining the transitions and the initial state. The initial state $\langle\{a_{init}\}, [\overline{a_{init}a_{init}} \mapsto \langle 0, false\rangle], \{\overline{a_{init}a_{init}}\}, \emptyset\rangle$ confirms the lemma. Now, consider the transitions:

—Transition $AC(a_i, a_j)$ creates a reference with $n = 0$ and makes the right side of the equation zero.

—Transitions $RC1(a_i, a_m, a_j)$, $RC1(a_i, a_j, a_m)$, $MM1(a_i)$ where $i = j$, $MS1(a_i, a_j)$, and $RC2(a_m, a_i, a_j)$ increase both sides of the equation by one.

—Transitions $RC2(a_i, a_j, a_m)$, $RC2(a_i, a_m, a_j)$, $RC3(a_i, a_j, a_m)$, $RC3(a_i, a_m, a_j)$, $RC3(a_m, a_i, a_j)$, and $MS2(a_i, a_j)$ convert a message to another message described in the right side of the equation, which does not change the equation.

—Transition $ACK(a_i, a_j)$ decreases both sides of the equation by one.

—Other transitions have no effect on the equation.

□

Lemma 5.12 says that if there exists $mr_x\langle a_i, a_m, a_j\rangle$, $mr_x\langle a_i, a_j, a_m\rangle$ (an in-transit message containing a reference), $mir_x\langle a_i, a_m, a_j\rangle$, $mir_x\langle a_i, a_j, a_m\rangle$, $mir_x\langle a_m, a_i, a_j\rangle$ (an in-transit inverse reference registration message), $m_x\langle a_i, a_j\rangle$ (an in-transit regular message), or $mack_x\langle a_i, a_j\rangle$ (an in-transit acknowledgement), then there exist a protected reference $\overline{a_i a_j}$ and a pseudo-root a_i.

LEMMA 5.12. *Existence of actor reference.*
Let the configuration of the actor system be $S = \langle A, R, IR, M\rangle$. *Let the expected acknowledgement counter equation be:*

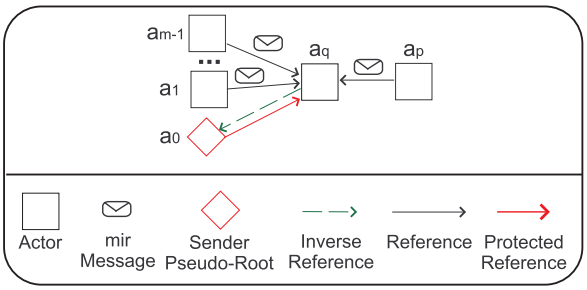

Fig. 6. Lemma 5.13: safe imprecise inverse references.

$$
\begin{aligned}
n \quad = \quad & |\{mr_x\langle a_i, a_m, a_j\rangle \mid mr_x\langle a_i, a_m, a_j\rangle \in M\}| \quad + \\
& |\{mr_x\langle a_i, a_j, a_m\rangle \mid mr_x\langle a_i, a_j, a_m\rangle \in M\}| \quad + \\
& |\{mir_x\langle a_i, a_m, a_j\rangle \mid mir_x\langle a_i, a_m, a_j\rangle \in M\}| \quad + \\
& |\{mir_x\langle a_i, a_j, a_m\rangle \mid mir_x\langle a_i, a_j, a_m\rangle \in M\}| \quad + \\
& |\{mir_x\langle a_m, a_i, a_j\rangle \mid mir_x\langle a_m, a_i, a_j\rangle \in M\}| \quad + \\
& |\{m_x\langle a_i, a_j\rangle \mid m_x\langle a_i, a_j\rangle \in M\}| \quad + \\
& |\{mack_x\langle a_i, a_j\rangle \mid mack_x\langle a_i, a_j\rangle \in M\}|
\end{aligned}
$$

Then the following statement is true:

$$n > 0 \Longrightarrow R(\overline{a_i a_j}) = \langle n, d\rangle.$$

Proof. The lemma can be proven by contradiction. Let the current time of the system be t_e. Let $\overline{a_i a_j} \notin dom(R)$ at time t_e. Since $n > 0$ at t_e, $\exists t_s : \overline{a_i a_j} \in dom(R)$ at time t_s and $\overline{a_i a_j} \notin dom(R)$ at time t_{s+1}, such that $t_s < t_e$ and t_s is the closest point to t_e (no re-creation allowed). Since the reference has been deleted and never re-created during t_s to t_e, $n = 0$ at t_s and t_{s+1}, implying no message of the right side of the equation is in transit. Consequently, $n = 0$ at t_e which contradicts the assumption. Therefore, $\overline{a_i a_j} \in dom(R)$ at time t_e. By Lemma 5.11, $R(\overline{a_i a_j}) = \langle n, d\rangle$ for some d and proves the lemma. □

From the perspective of the safety property of a reference listing algorithm, a referenced actor, namely a_q, must have at least one inverse reference to figure out it is referenced. The pseudo-root approach guarantees Lemma 5.13, which is used to prove the safety property (Theorem 5.14). Then we will use Lemma 5.10 to prove the liveness property (Theorem 5.15). Figure 6 illustrates Lemma 5.13.

Lemma 5.13. *Safe imprecise inverse references.*
Let $S = \langle A, R, IR, M\rangle$ be an actor system configuration and $\{a_p, a_q\} \subseteq A$. The following statement is true at any time:

$$
\begin{gathered}
\overline{a_p a_q} \in URS(S, a_q) \Longrightarrow \\
\exists a_0, a_1, ..., a_m : \overline{a_0 a_q} \in PRS(S, a_q) \wedge \\
(R(\overline{a_0 a_q}) = \langle n_0, d\rangle \wedge n_0 > 0) \wedge \\
(mir_{xi}\langle a_i, a_{i+1}, a_q\rangle \in M \ \textit{where}\ i = 0 \ \textit{to}\ m-1 \ \textit{and}\ a_m = a_p).
\end{gathered}
$$

Proof. To prove the lemma by induction, we need to build a partial order of reference creation. The first references to Actor a_q are either created by the actor

creation transition or at the initial state of the actor system because Actor a_q is the initial actor. With the existence of the first references to a_q, they can be duplicated by the reference passing transition $RC1$. Then the n_{th} $RC2$ transition of a_q, $RC2(a_{w1}, a_{w2}, a_q)$, creates the n_{th} reference to a_q, where $\{a_{w1}, a_{w2}\} \subseteq A$.

Basis: Prove the lemma for $n = 1$ (the first references).
First, the initial configuration of an actor system does not have any unpaired references. Second, the actor creation transition does not create unpaired references. Therefore, the basis is true.

Induction step: Assume $\exists k : \forall n \leq k$ the lemma is true. Now, consider a $(k+1)_{th}$ reference in any actor system.
Let $RC1(a_u, a_p, a_q)$ be the transition to initialize reference creation of $\overline{a_p a_q}$, and $RC2(a_u, a_p, a_q)$ be the transition to create $\overline{a_p a_q}$. That is, $\exists S_s, S_1, S_2, S_3 : S_s \xrightarrow{RC1(a_u,a_p,a_q)} S_1 \rightarrow^* S_2 \xrightarrow{RC2(a_u,a_p,a_q)} S_3 \rightarrow^* S$. Let the corresponding time of S_s, S_1, S_2, S_3, and S be t_s, t_1, t_2, t_3, and t. Now, consider $\overline{a_u a_q}$ at t_s, a reference whose order is less than $k+1$. There are two possible cases:

—$\overline{a_u a_q} \in PRS(S_s, a_q)$ at t_s: Message $mr_{x1}\langle a_u, a_p, a_q\rangle$ created by Transition $RC1(a_u, a_p, a_q)$ will not be consumed until the system transits to S_3. Let $R(\overline{a_u a_q}) = \langle n_{u,q}, d\rangle$. According to Lemma 5.11, $n_{u,q} \geq 0$ between t_s and t_2, which also means $RD1(a_u, a_q)$ cannot happen. $S_2 \xrightarrow{RC2(a_u,a_p,a_q)} S_3$ makes Message $mr_{x1}\langle a_u, a_p, a_q\rangle$ to be consumed and then Message $mir_{x2}\langle a_u, a_p, a_q\rangle$ to be produced, which means $n_{u,q} \geq 1$ at t_3. We know $\overline{a_p a_q} \in URS(S, a_q)$ at t, which means $mir_{x2}\langle a_u, a_p, a_q\rangle$ is still at t, implying $n_{u,q} \geq 1$ by Lemma 5.11. Therefore, the following property is guaranteed at t: $\overline{a_u a_q} \in PRS(S, a_q) \wedge (R(\overline{a_u a_q}) = \langle n_{u,q}, d\rangle \wedge n_{u,q} > 0) \wedge (mir_{x2}\langle a_u, a_p, a_q\rangle) \in M$.

—$\overline{a_u a_q} \in URS(S_s, a_q)$ at t_s: There are two sub-cases:
Sub-Case 1: Assume $(mir_x\langle a_u, a_p, a_q\rangle)$ is not consumed before t, and let $R(\overline{a_u a_q}) = \langle n_{u,q}, d\rangle$. According to Lemma 5.12, $n_{u,q} > 0$, and thus $\overline{a_u a_q}$ is at t. According to the induction assumption, $\exists a_0, a_1, ..., a_m : \overline{a_0 a_q} \in PRS(S, a_q) \wedge (R(\overline{a_0 a_q}) = \langle n_0, d\rangle \wedge n_0 > 0) \wedge (mir_{xi}\langle a_i, a_{i+1}, a_q\rangle \in M$ where $i = 0$ to $m-1$ and $a_m = a_u$). We know $\overline{a_p a_q} \in URS(S, a_q)$ at t, which means $mir_{x2}\langle a_u, a_p, a_q\rangle$ is at t and proves this sub-case.
Sub-Case 2: Assume $(mir_x\langle a_u, a_p, a_q\rangle)$ has been consumed between t_3 and t, which means $\forall S_x : \overline{a_u a_q} \in PRS(S_x, a_q)$ where S_x is an actor system configuration existing on a time point from t_3 to t. Let $R(\overline{a_u a_q}) = \langle n_{u,q}, d\rangle$. Since $\overline{a_p a_q} \in URS(S, a_q)$ is at t, $mir_{x2}\langle a_u, a_p, a_q\rangle$ is at t. By Lemma 5.11, $n_{u,q} > 0$. By Lemma 5.12, $R(\overline{a_u a_q}) = \langle n_{u,q}, d\rangle$. Therefore, the following property is true at t: $\overline{a_u a_q} \in PRS(S, a_q) \wedge (R(\overline{a_u a_q}) = \langle n_{u,q}, d\rangle \wedge n_{u,q} > 0) \wedge mir_{x2}\langle a_u, a_p, a_q\rangle \in M$.

To conclude, the lemma is true by induction.

□

THEOREM 5.14. *Safety property.*
Let $S = \langle A, R, IR, M\rangle$ be an actor system configuration and $\{a_p, a_q\} \subseteq A$. Let the current time be t. The following statement is always true:

$$\overline{a_p a_q} \in dom(R) \Longrightarrow \exists a_u : \overline{a_u a_q} \in IR$$

PROOF. If $\overline{a_p a_q} \in URS(S, a_q)$ at t is true, $\exists a_0 : \overline{a_0 a_q} \in PRS(S, a_q)$ is also true by Lemma 5.13, implying $\overline{a_0 a_q} \in IR$. Otherwise, $\overline{a_p a_q} \in PRS(S, a_q)$ at t must be true, implying $\overline{a_p a_q} \in IR$ is true. Both cases confirm the theorem. □

THEOREM 5.15. *Liveness property.*
Let $S = \langle A, R, IR, M\rangle$ be an actor system configuration. Let $a_q \in A$ such that a_q is not a pseudo-root actor at current time point t. The following statement is true at any time:

$$\nexists a_p : R(\overline{a_p a_q}) = \langle n_p, d_p\rangle \text{ at } S \Longrightarrow \exists S_{future} : S \rightarrow^* S_{future} \wedge (|UIRS(S_{future}, a_q)| = 0 \text{ at } S_{future}).$$

PROOF. Let $S \rightarrow^* S_{future}$ where the time point of S_{future} is t_{future}. Since a_q is not a pseudo-root actor, it cannot execute any spontaneous transition. Thus $\nexists a_p : \overline{a_p a_q} \in dom(R)$ is always true for any state after S. By Lemma 5.10, $|UIRS(S, a_q)| = |MDS(S, a_q)|$ at t. Thus there exist $|UIRS(S, a_q)|$ inverse reference deletion messages of a_q, which will trigger $|UIRS(S, a_q)|$ inverse reference deletion transitions of a_q. With the fairness assumption of the actor model of computation, all those inverse reference deletion transitions will eventually occur at a future state S_{future}, making $|UIRS(S_{future}, a_q)| = 0$. □

5.3 The Pseudo-Root Approach Properties in a Distributed Environment

The pseudo-root approach can be used to support local marking in a distributed environment, that is, performing actor marking in a *partial view* of an actor system to identify all local garbage. With the concept of the partial view, we can define the global pseudo-root actors which are treated as roots. The pseudo-root approach guarantees one-step back-tracing safety (Theorem 5.18). It says that if an actor in a partial view is referenced by another actor outside the partial view, the actor is either a global pseudo-root actor or is directly reachable from a sender pseudo-root actor in the partial view (see Figure 7).

DEFINITION 5.16. *Partial view of an actor system.*
Let $S = \langle A, R, IR, M\rangle$, and $V = \langle A', R', IR'\rangle$. V is a partial view of S if and only if

. $A' \subseteq A \wedge$
. $(\forall a_p : (a_p \in A \wedge \overline{a_p a_q} \in dom(R)) \Longrightarrow \overline{a_p a_q} \in dom(R')) \wedge$
. $(\forall a_q : a_q \in A \wedge \overline{a_p a_q} \in IR \Longrightarrow \overline{a_p a_q} \in IR')$.

DEFINITION 5.17. *Global pseudo-root actor.*
Let V be a partial view of S, where $V = \langle A', R', IR'\rangle$. Actor a_q is a global pseudo-root actor of V if and only if

$$\exists a_p : a_p \notin A' \wedge a_q \in A' \wedge \overline{a_p a_q} \in IR'.$$

THEOREM 5.18. *One-step back-tracing safety.*
Let V be a partial view of S, where $V = \langle A', R', IR'\rangle$ and $S = \langle A, R, IR, M\rangle$.

$$\exists a_p, a_q : a_p \notin A' \wedge a_q \in A' \wedge R(\overline{a_p a_q}) = \langle n, d\rangle \Longrightarrow \exists a_w : (\overline{a_w a_q} \in IR' \wedge a_w \notin A') \vee (R'(\overline{a_w a_q}) = \langle n', d'\rangle \wedge n' > 0 \wedge a_w \in A').$$

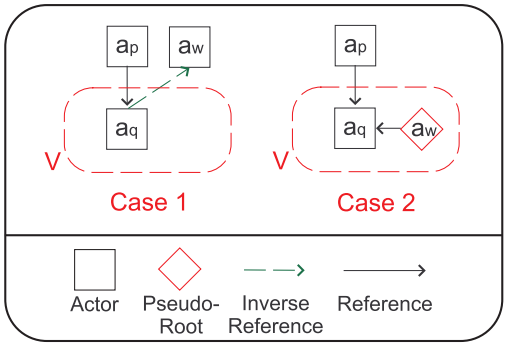

Fig. 7. Theorem 5.18: one-step back-tracing safety.

PROOF. Let the current time be t. Now, consider the following two cases:

—*Case 1:* Let $\overline{a_p a_q} \in PRS(S, a_q)$ at t, which means $(\overline{a_p a_q} \in IR)$. $a_q \in A'$ implies $(\overline{a_p a_q} \in IR')$.

—*Case 2:* Let $\overline{a_p a_q} \in URS(S, a_q)$ at t. By Lemma 5.13, $\exists a_0 : \overline{a_0 a_q} \in PRS(S, a_q) \wedge (\exists n_0 : R(\overline{a_0 a_q}) = \langle n_0, d\rangle \wedge n_0 > 0)$. Now consider two sub-cases. First, $a_0 \in A'$ implies $(R'(\overline{a_0 a_q}) = \langle n_0, d\rangle \wedge n_0 > 0)$. Second, $a_0 \notin A'$ implies $\overline{a_0 a_q} \in IR'$.

Therefore, the theorem is true. □

5.4 Actor Garbage Collection without the Live Unblocked Actor Principle

The live unblocked actor principle is not always true in actor systems. The pseudo-root approach also supports actor marking algorithms in this context. In a local host, we re-define sender pseudo-root actors as unblocked actors. Global pseudo-root actors are re-defined as global roots, consisting of the remotely referenced global roots and the unblocked-reachable global roots. Remotely referenced global roots are defined in Definition 5.17. Theorem 5.19 is the safety property of local actor marking. It says that if an actor is remotely referenced, it either has an inverse reference to figure out it is remotely referenced (the remotely referenced global root), or it is reachable from an unblocked actor and the unblocked actor can reach an unblocked-reachable global root (see Figure 8).

THEOREM 5.19. *One-step back-tracing and forward validating safety. Let V be a partial view of S, where $V = \langle A', R', IR'\rangle$ and $S = \langle A, R, IR, M\rangle$.*

$$\exists a_p, a_q : a_p \notin A' \wedge a_q \in A' \wedge R(\overline{a_p a_q}) = \langle n, d\rangle \Longrightarrow$$
$$\exists a_0 : (\overline{a_0 a_q} \in IR' \wedge a_0 \notin A') \vee$$
$$(\exists a_1, a_2, ..., a_u : R'(\overline{a_0 a_q}) = \langle n_0, d_0\rangle \wedge n_0 > 0 \wedge a_0 \in A'$$
$$\wedge\ R'(\overline{a_i a_{i+1}}) = \langle n_i, d_i\rangle \wedge a_u \notin A' \text{ where } i = 1 \text{ to } u - 1).$$

PROOF. Let us consider two cases:

—*Case 1:* Let $\overline{a_p a_q} \in PRS(S, a_q)$, which means $(\overline{a_p a_q} \in IR)$. $a_q \in A'$ implies $(\overline{a_p a_q} \in IR')$.

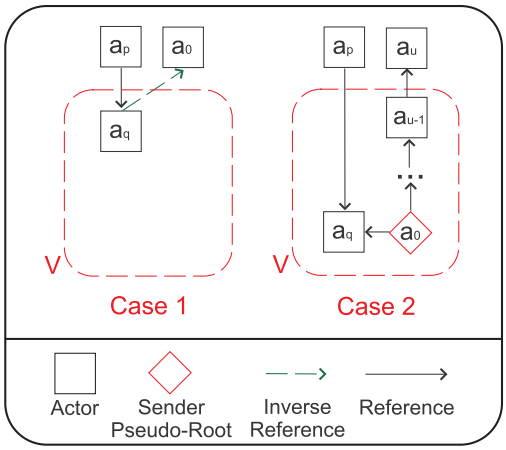

Fig. 8. Theorem 5.19: one-step back-tracing and forward validating safety.

—*Case 2:* Let $\overline{a_p a_q} \in URS(S, a_q)$. By Lemma 5.13, $\exists a_0, a_1, ..., a_m : \overline{a_0 a_q} \in PRS(S, a_q) \wedge (R(\overline{a_0 a_q}) = \langle n_0, d\rangle \wedge n_0 > 0) \wedge (mir_{xi}\langle a_i, a_{i+1}, a_q\rangle \in M$ where $i = 0$ to $m-1$ and $a_m = a_p)$, which implies $(R(\overline{a_i a_{i+1}}) = \langle n_i, d_i\rangle \wedge n_i > 0$ where $i = 1$ to $m-1$ and $a_m = a_p)$ by Lemma 5.12. Now consider the following two sub-cases.
First, let $a_0 \in A'$, which implies $R'(\overline{a_0 a_q}) = \langle n_0, d\rangle \wedge n_0 > 0$. Let $A_{0\_to\_u-1} = \{a_0, a_1, ..., au-1\} \subseteq A'$. Notice that $m \geq u \geq 0$ because $(a_0 \in A' \wedge a_m \notin A')$. $R(\overline{a_i a_{i+1}}) = \langle n_i, d_i\rangle \wedge n_i > 0$ where $i = 1$ to $m-1$ and $a_m = a_p)$ and $(A_{0\_to\_u-1} \subseteq A')$ implies $(R'(\overline{a_i a_{i+1}}) = \langle n_i, d_i\rangle \wedge a_u \notin A'$ where $i = 1$ to $u-1)$. Thus the theorem is true for this sub-case.
Second, $a_0 \notin A'$ implies $\overline{a_0 a_q} \in IR'$.

Therefore, the theorem is true. □

6. EXPERIMENTAL RESULTS

The pseudo-root approach is implemented and used as the core garbage collection mechanism for SALSA programs [Worldwide Computing Laboratory 2007]. *SALSA* (*Simple Actor Language, System and Architecture*) is an actor-oriented programming language designed and implemented to introduce the benefits of the actor model while keeping the advantages of object-oriented programming [Varela and Agha 2001]. Abstractions include actors (active objects), asynchronous message passing, universal naming, migration, and advanced coordination constructs for concurrency. SALSA is compiled into Java and hence preserves many of Java's useful object oriented concepts — mainly, encapsulation, inheritance, and polymorphism. Because each actor maintains an encapsulated state, SALSA uses strict pass-by-value communication [3]. SALSA abstractions enable the development of

[3]Actor references (names) are first-class and therefore they can also be passed by value.

dynamically reconfigurable applications. A SALSA program consists of universal actors that can migrate to other computing nodes at run-time.

SALSA provides actor garbage collection, which identifies actor garbage and let the Java garbage collector reclaim it. Intra-actor garbage produced by updating references (the assignment operation) is handled by the Java garbage collector as well. In this section, we will use several types of applications to measure the performance of our implementation.

Actor GC Mechanism to Measure

To understand the impact of actor garbage collection, we measure actor garbage collection using three different mechanisms: *No-GC*, *GDP*, and *LGC*. By using these mechanisms, we can understand the overhead each actor garbage collection algorithm imposes on the actor system. The mechanisms are described as follows:

—*No-GC*: Data structures and algorithms for actor garbage collection are not used.
—*GDP*: The local garbage collector is not activated. Only the *garbage detection protocol* (the implementation of the pseudo-root approach) is used.
—*LGC*: The local garbage collector is activated every n seconds or in the case of insufficient memory (n=2 for the tests in this section).

6.1 Overhead Breakdown

There are totally four kinds of mutation operations in actor garbage collection: *actor creation*, *actor reference passing*, *actor reference deletion*, *and actor migration*. To understand if the actor garbage collection mechanism is intrusive or not, we also measure the effect on loops containing multiplication and division operations. Since the extra execution time required for actor reference deletion and actor reclamation is handled by Java virtual machines, we do not provide any overhead breakdown for them. However, their overheads should be proportional to the overhead of actor creation because the Java garbage collector needs to reclaim extra objects [4] produced by actor creation. All testing applications used in this section were developed using *SALSA 1.1.1*.

Overhead Breakdown of Local Actor GC in a Uniprocessor Environment. The experiments were performed on a Dell Inspiron 600m laptop, equipped with a 1.6 GHz Intel Pentium M processor and 512 MB of RAM. The operating system used was Microsoft Windows XP, and the Java VM was Java HotSpot(TM) Client VM (build 1.5.0_06-b05, mixed mode). Experimental results are shown in Tables I, II, III, and IV.

Table I shows the overhead of actor creation. The overhead is relatively huge while creating a small actor because each actor maintains a data structure for the actor referential relationship. The ratio of the extra data structure for actor garbage collection goes down while the size of a newly created actor increases. Table II shows the overhead of message passing. Table III shows the overhead of reference passing. The overhead is almost 100% for local actor garbage collection because each reference passing event generates at least 3 extra messages. Table IV shows the overhead of a regular computation, represented by the multiplication-division

[4]The total size of the required objects is fixed in the experiments.

Table I. Actor creation (ms) and its overhead (%) in a uniprocessor environment.

Actor Creation	No-GC	GDP	GDP+ LGC	GDP Overhead	GDP+LGC Overhead
Empty Actor	0.872	1.001	1.382	15%	58%
1 KB State	1.412	1.423	1.743	1%	23%
10 KB State	1.873	1.913	2.443	6%	30%

Table II. Message passing (μs) and its overhead (%) in a uniprocessor environment.

Message Passing	No-GC	GDP	GDP+ LGC	GDP Overhead	GDP+LGC Overhead
Empty Parameters	6.97	7.13	7.212	2%	3%
1 KB Parameters	57.68	59.08	59.68	2%	3%
10 KB Parameters	59.3	59.28	60.7	0%	2%

Table III. Reference passing (ms) and its overhead (%) in a uniprocessor environment.

Reference Passing	No-GC	GDP	GDP+ LGC	GDP Overhead	GDP+LGC Overhead
1 Reference	0.549	1.094	1.118	99%	104%
10 References	0.577	1.156	1.202	94%	102%
100 References	0.691	1.348	1.384	95%	100%

computation whose overhead is negligible. The minus-overhead could be attributed to the behavior of the operating system or the Java VM. A -1% overhead should be considered as a precision problem of measurement.

Overhead Breakdown of Local Actor GC in a Concurrent Environment. We used the same mechanisms and applications described above to measure the overhead of local actor garbage collection in a concurrent environment. The experiments were performed on an IBM pSeries 655 machine, equipped with four 1.7 GHz IBM POWER4 processors and 4 GB of RAM. The operating system used was IBM AIX 5.3, and the Java VM was Classic VM (build 1.4.2, J2RE 1.4.2 IBM AIX 5L for PowerPC). Experimental results are shown in Tables V, VI, and VII. We decided not to include the experimental results of the multiplication-division (ms) and its overhead because it highly depends on the scheduling policy of the operating system and the Java VM — in some cases the system with actor garbage collection can outperform the system without actor garbage collection by almost 100%, while in some cases the system with actor garbage collection can have more than 50% overhead.

The actor garbage collection overhead in the concurrent system shows a significant improvement compared to the uniprocessor environment, shown in Tables V, VI, and VII. We attribute it to the scheduling policy of the native operating system — the operating system wisely uses another idle processor for concurrent actor garbage collection, which avoids CPU time contention with users' applications. The minus-overhead in Table V shows that the scheduling policy of the operating system (or the Java VM) can have high influence on the performance of concurrent applications.

Overhead Breakdown of Actor GC in a Cluster Environment. The overhead of actor migration, message passing, and reference passing are evaluated in this sub-

Table IV. Multiplication-division (Secs) and its overhead (%) in a uniprocessor environment, where each actor performs several loops, each of which contains a double-precision multiplication operation and a double-precision division operation.

Mul-Div	No-GC	GDP	GDP+ LGC	GDP Overhead	GDP+LGC Overhead
3×10^8 Loops × 1 Actor	7.971	7.971	8.181	0%	3%
3×10^7 Loops × 10 Actors	7.971	7.972	8.116	0%	2%
3×10^6 Loops × 100 Actors	8.052	8.082	8.146	0%	1%
3×10^5 Loops × 1000 Actors	8.692	8.603	8.933	-1%	3%

Table V. Actor creation (ms) and its overhead (%) in a quad-core processor environment.

Actor Creation	No-GC	GDP	GDP+ LGC	GDP Overhead	GDP+LGC Overhead
Empty Actor	7.594	6.222	6.43	-18%	-15%
1 KB State	4.611	4.717	5.191	2%	13%
10 KB State	4.446	5.015	5.442	13%	22%

Table VI. Message passing (μs) and its overhead (%) in a quad-core processor environment.

Message Passing	No-GC	GDP	GDP+ LGC	GDP Overhead	GDP+LGC Overhead
Empty Parameters	9.914	9.924	9.948	0%	0%
1 KB Parameters	86.6	85.73	87.64	-1%	1%
10 KB Parameters	86.74	86.15	86.44	-1%	0%

section. We used two IBM pSeries 655 machines (intra-cluster) to measure the overhead of actor garbage collection. Experiments results are shown in Tables VIII, IX, and X. The major overhead still comes from reference passing, on average 18% (Table X). The overhead shown in Tables VIII and IX is negligible.

6.2 Scalability Test

In this subsection, we evaluate the overhead using a scalable application, namely the *maximum likelihood evaluation fitter* (*MLE fitter*), to evaluate a large set of data, where *likelihood* is defined as the product of the probabilities of observing each event (the input data set) given a set of fit parameters [Wang et al. 2006].

The Maximum Likelihood Evaluation (MLE) Fitter. According to particle physics, *particles* which make up our universe have wave-like behavior and thus their identities and properties can be determined by *partial wave analysis (PWA)* [Cummings and Weygand 2003]. To discover the identities and properties of particles, and the forces and interactions between these particles, scientists use a particle accelerator to create a high energy collision of particles. The collision may produce a spray of particles. Some of the particles can live long enough to be observed, while some

Table VII. Reference passing (ms) and its overhead (%) in a quad-core processor environment.

Reference Passing	No-GC	GDP	GDP+ LGC	GDP Overhead	GDP+LGC Overhead
1 Reference	0.818	0.948	0.957	16%	17%
10 References	0.852	0.985	0.993	16%	17%
100 References	0.929	1.067	1.113	15%	20%

Table VIII. Actor migration (ms) and its overhead (%) in a distributed environment.

Actor Creation	No-GC	GDP	GDP+ LGC	GDP Overhead	GDP+LGC Overhead
Empty Actor	158.875	157.755	158.7	-1%	0%
1 KB State	160.005	159.15	158.91	-1%	-1%
10 KB State	159.855	159.605	159.095	0%	0%

Table IX. Message passing (ms) and its overhead (%) in a distributed environment.

Message Passing	No-GC	GDP	GDP+ LGC	GDP Overhead	GDP+LGC Overhead
Empty Parameters	0.149	0.149	0.149	0%	0%
1 KB Parameters	0.251	0.251	0.251	0%	0%
10 KB Parameters	0.336	0.337	0.336	0%	0%

may decay [5] into other kinds of particles after an extremely short time, making them impossible to be observed. The existence of the short lived particles can only be inferred from correlations in the final state particles into which they decay. There are many ways of reaching the final system through various intermediate states, and each possibility must be considered. By varying the amount of each intermediate state to fit the observed final state, PWA can determine the identities of the short-lived particles.

The purpose of the fits is to find the most probable intermediate states. The fits are done using *maximum likelihood evaluation (MLE)*. The likelihood is defined as the product of the probabilities of observing each event given a set of fit parameters. In practice, people usually use the negative logarithm likelihood to find the minimum value, which represents the maximum likelihood. In our case, the equation is described as:

$$-\ln(\mathcal{L}) = -\sum_{i}^{n} \ln\left(\left|\psi_{\alpha}^{p}\psi_{\alpha}^{d}(\tau_i)\right|^2\right) - n\psi_{\alpha}^{p}\Psi_{\alpha\alpha'}\psi_{\alpha'}^{p*} \quad (1)$$

where the sum over i runs over all events in the data set, and the sums over the repeated αs, the fit parameter index, are implicit. The ψ_{α}^{p} are the complex fit parameters, related to the amount of the intermediate state α produced. The $\psi_{\alpha}^{d}(\tau_i)$ is the identity (*quantum amplitude*) for the i^{th} event with angles τ_i assuming intermediate state α. The possibility of non-interfering data has been ignored here. While physically important, it is a detail which further complicates the expression for the likelihood, yet serves no illustrative purpose for the discussion at hand. The

[5]Particle decay refers to the transformation of a fundamental particle into other fundamental particles.

Table X. Reference passing (ms) and its overhead (%) in a distributed environment.

Reference Passing	No-GC	GDP	GDP+ LGC	GDP Overhead	GDP+LGC Overhead
1 Reference	1.063	1.237	1.236	16%	16%
10 References	1.152	1.322	1.326	15%	15%
100 References	1.221	1.398	1.409	15%	15%

second term on the right hand side is the normalization integral, where any known inefficiencies of the detector are taken into account. The total number of events in the data being fit is n; and $\Psi_{\alpha\alpha'}$ is the result of the normalization integral, done numerically before the fit is performed.

We used the *simplex* algorithm, which finds the most probable fit and iteratively improves the output, in our implementation of the MLE fitter. Finding the best fit parameters to a typical data set requires a given set of initial fit parameters and hundreds or even thousands of trials. In our case the MLE fitter evaluates the maximum likelihood of a given set of complex amplitudes and the observed events from collisions of particles. The execution time of the MLE fitter can be improved by using distributed computing if calculating the summation term of the negative logarithm likelihood equation requires a long time to finish. For example, if n in Equation 1 is large enough ($> 10^5$).

Results. The test was performed in a cluster consisting of three 4-dual-core 2.2 GHz Opteron machines (8 processors each) with 32 GB of RAM and twelve 4-single-core 2.2 GHz Opteron machines with 16 GB of RAM. The operating system used was Linux 2.6.15, and the Java VM was Java HotSpot Server VM (build 1.5.0_08-b03, mixed mode).

We used a set of 9053×2^6 events and 7 complex number parameters as the input for the MLE fitter. The MLE fitter used a static load balancing approach to distribute data to each *theater* [6], where for each processor a theater is started to host actors. The MLE fitter is sensitive to any delay at any theater because the execution time per fit function call is the longest execution time per fit function call among all the participating theaters. Our experiments show that the overall overhead of our implementations is on average 11% for 16 or more theaters, and on average 6% for 8 or less theaters (see Figures 9 and 10). The MLE fitter running on 64 or more theaters has relatively bad performance, which can be attributed to the execution time per function call being down to 1 second, making it relatively sensitive to any kind of interruption.

7. RELATED WORK

In this section we introduce related work, including reference counting/listing algorithms, indirect/hybrid algorithms, and actor garbage collection algorithms.

7.1 Reference Counting/Listing

Distributed reference counting (or listing) algorithms, also known as distributed reference counting protocols, are inherently incremental and thus they do not re-

[6] A theater can be thought of as a virtual machine for the SALSA programming language, providing support for distributed computing.

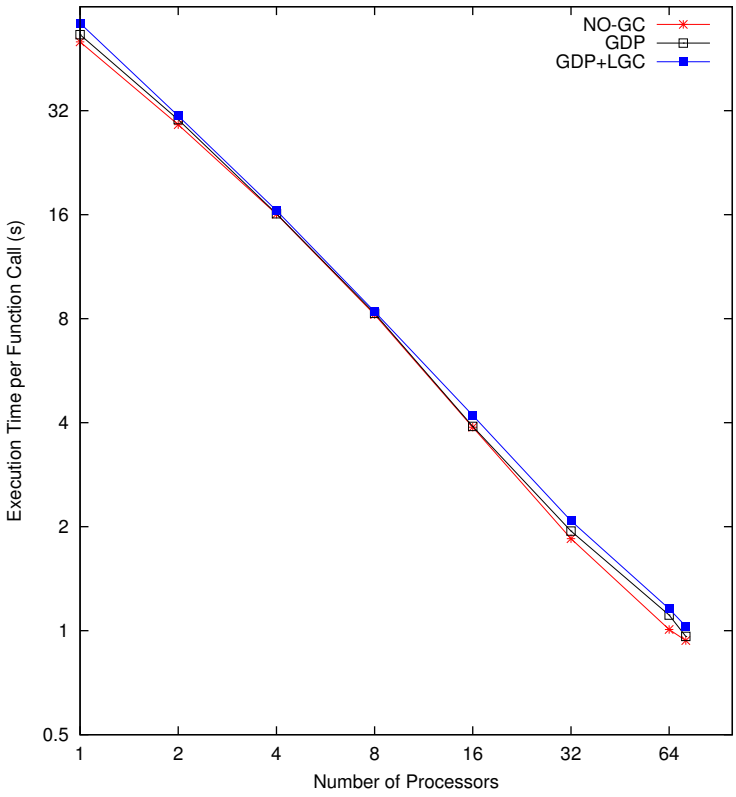

Fig. 9. Execution time per fit function call vs. the total number of processors. Four kinds of mechanisms are used to evaluate the implementation of our actor garbage collection algorithms.

quire complex synchronization for garbage collection. Typically, they require that either a referenced object has to maintain a counter, or a reference table has to keep track of every incoming/outgoing inter-node reference. Whenever an object is referenced, its counter has to increment by one (or creates an inverse reference in the reference table). A common disadvantage of them is that they cannot reclaim cyclic garbage (mutually referenced objects) unless a hybrid strategy is used. These reference counting algorithms are similar to the pseudo-root approach but they tend to be more synchronous — all of them rely on First-In-First-Out (FIFO) communication or timestamp-based FIFO (simulated FIFO) communication in the application level, and some of them are even totally synchronous by using remote-procedure-call. Since actor communication is defined as asynchronous, unordered, and message-driven, these algorithms cannot be reused directly by actor systems.

The Lermen-Maurer protocol [Lermen and Maurer 1986] is the first distributed reference counting algorithm. It requires three messages per reference duplication, and one for reference deletion. The application message with the reference to pass and the increment system message to the object owner process are sent simultaneously. To avoid premature reference deletion, the object owner process must send

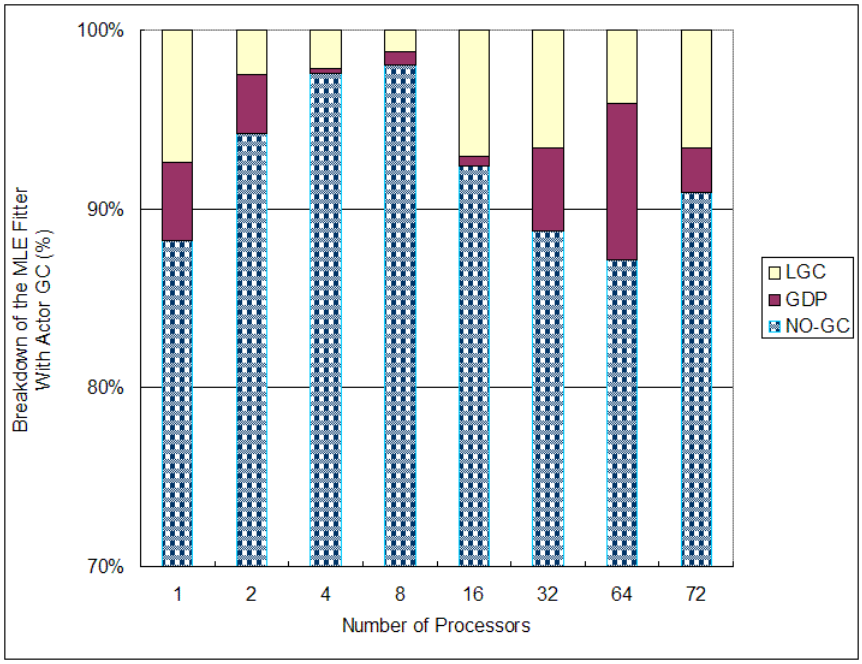

Fig. 10. Breakdown of the actor garbage collection mechanism.

an acknowledgement to the application message receiver process. References can only be deleted if the total number of received acknowledgements for the reference is equal to the total number of receipts of the reference from application message passing. The Lermen-Maurer protocol requires FIFO communication.

The weighted reference counting protocol [Bevan 1987; Watson and Watson 1987] avoids sending increment messages, and only one decrement message is required for reference deletion in most cases. The algorithm requires each object and each reference maintains its *weight*. The weight of an object is always equal to the sum of the total weights of the references to the object. A new object and the original reference to it are created with a predefined maximum weight. When a reference is duplicated, the weight of the reference is divided into two positive new values, and the sum of the new values is equal to the original weight of the reference. One value is assigned to the original reference and the other one to the new reference. While a reference is deleted, a decrement message containing the weight of the reference is sent to the object which it is pointing to. An object is deleted safely when its weight drops to zero. However, an alternative approach must be used when an object with its weight equal to 1 is going to duplicate, such as creating an indirect object with a value of max weight and then dividing the value by two to the original reference and the copied reference. Corporaal et al in 1990 [Corporaal et al. 1990] use a similar approach by extra reference tables instead of indirect objects.

The indirect reference counting algorithm, proposed by Piquer [Piquer 1991],

avoids sending increment messages in order to optimize the number of extra messages to send. The algorithm maintains a special data structure for each reference which includes: 1) the number of duplications, 2) the parent pointer, and 3) the reference. The number of copies and the parent pointer are used to form an inverted diffusion tree to present the history of reference duplications. The root is the process that owns the object. The parent pointer is not used to point to the parent object, but for the history of reference duplication. Only the leaf references of the inverted diffusion tree can be deleted because the algorithm has to maintain the inverted diffusion tree structure. Object migration is done by changing the parent process to the migration destination. Deletion of references may produce extra messages. The algorithm may preserve zombie references because they are not leaf references.

The SSPC (Stub-Scion Pair Chains) protocol [Shapiro 1991; Shapiro et al. 1992] is a reference listing protocol which uses the techniques of timestamps to ensure FIFO communication. If the FIFO order is violated by comparing the timestamps of messages, a redelivery is occurred to maintain the FIFO order. Synchronous communication of users' applications is assumed, but the protocol itself is asynchronous. Whenever a reference is copied, an intermediate stub-scion pair is created to form an indirect path to connect the reference receiver process to the reference owner process. The SSPC protocol supports object migration by leaving an intermediate pointer at the original space to the newly migrated object. Another background protocol can be applied to reduce extra stub-scion pairs in a reference path to improve message delivery performance and robustness.

Birrell's protocol [Birrell et al. 1993] is attributed as a remote-procedure-call, reference listing algorithm, and thus it is a synchronous algorithm. It is used in distributed acyclic garbage collection of Java RMI. The algorithm relies on sequential remote procedure calls. Upon reference duplication, the original call with reference parameters to the reference receiver process is made, following by another dirty call to the object owner process. Reference deletion is implemented as a clean call from the reference holder process to the object owner process. Moreau et al. [Moreau et al. 2005] formalize the algorithm and present with correctness proofs, and also extend the algorithm with non-FIFO communication semantics.

Moreau's protocol [Moreau 2001] is a reference listing protocol which requires point-to-point FIFO communication. Each process maintains a table for counting the references it held, and each object maintains a counter for counting the number of references to it. Once a reference is passed to another process, the original process (sender) should increase the counter of the reference and then sends the message out. If a process receives a message with a reference, it sends a special message *INC-DEC* to the process that the referenced object resides in. Upon receiving the INC-DEC message, a process increases the counter of the referenced object and sends a *DEC* message with the identity of the referenced object to the original process (sender). A process should decrease the counter of the corresponding reference of the referenced object. Deleting references is similar to the Lermen-Maurer protocol.

7.2 Distributed Trace-Based/Hybrid Algorithms

There are various trace-based/hybrid distributed garbage collection algorithms for passive object systems, where "hybrid" refers to the approach combining reference

counting/listing and trace-based strategies. The most important feature of these algorithms is that they collect at least some distributed cyclic garbage. Hughes' algorithm [Hughes 1985] uses global timestamp propagation from roots which requires long pause time for garbage collection and is very sensitive to failures. Liskov et al. [Liskov and Ladin 1986; Ladin and Liskov 1992] present a centralized algorithm which requires every client (local collector) to report inter-node references to a server. Vestal [Vestal 1987] assumes assistance from acyclic reference counting and tries to virtually delete a reference to see whether or not an object is garbage, which cannot detect all cycles. Maheshwari et al. [Maheshwari and Liskov 1995; 1997] and Le Fessant [Le Fessant 2001] propose heuristics-based algorithms which use the minimal number of inter-node references from a root to suspect some objects as garbage and then verifies the suspects. Lang et al. [Lang et al. 1992] propose a trace-based algorithm with the help of a reference listing protocol and local garbage collection to collect garbage hierarchically, which does not support migration and must stop the mutators while garbage collection is performing. Rodrigues et al. [Rodrigues and Jones 1996] present a dynamically partitioning approach which starts from suspecting an object as garbage, traces from it, and then forms a group for global garbage collection. During global garbage collection, mutators must be suspended for local live object marking. Overlapped partition can occur; it either causes deadlocks or no work can be done. The algorithm proposed by Veiga et al. [Veiga and Ferreira 2005] is also heuristics-based, and it uses asynchronous local snapshots to identify global garbage. Any change to the snapshots has to be updated by local mutators, forcing current global garbage collection to quit. Hudson et al. [Hudson et al. 1997] propose a generational collector where the address space of each computing node is divided into several disjoint blocks (cars), and cars are grouped together into several distributed trains. Each train represents a generation of objects, and forms a ring structure for distributed management. Objects can only move from an older generation car to a younger generation car, and the oldest car is eventually inspected. A car/train can be disposed of if there are no incoming inter-car/inter-train references to it. Blackburn et al. [Blackburn et al. 2001] suggest a methodology to derive a distributed garbage collection algorithm from an existing distributed termination detection algorithm [Francez 1980; Dijkstra and Scholten 1980; Matocha and Camp 1998], in which the distributed garbage collection algorithm developers must design another algorithm to guarantee a consistent global state. All of the above algorithms cannot be reused directly in actor systems because actors and passive objects are different in nature.

7.3 Actor Garbage Collection

The definition of garbage actors is different from the perspective of passive object garbage collection. For instance, the marking phase of passive object garbage collection has only two choices, the depth-first-search and the breadth-first-search. Marking algorithms for actor garbage collection are relatively various, including Push-Pull, Is-Black by Kafura et al. [Kafura et al. 1990], Dickman's graph partition merging algorithm by Dickman [Dickman 1996], and the actor transformation algorithm by Vardhan and Agha [Vardhan 1998; Vardhan and Agha 2002].

Most distributed actor garbage collection algorithms are snapshot-based due to the autonomous nature and the asynchronous communication of actors. The

global `Push-Pull` algorithm proposed by Kafura et al. [Kafura et al. 1995] uses the Chandy-Lamport snapshot algorithm [Chandy and Lamport 1985] to determine a precise global state, which is expensive and requires FIFO communication to flush communication channels. Garbage collection is hierarchical, consisting of local garbage collection and global garbage collection. Global roots are defined as actors with incoming remote references (remote inverse references) or outgoing remote references. Treating all actors with outgoing remote references as global roots is over-conservative because not every actor with outgoing remote references is live — only those reachable from an unblocked actor and have outgoing remote references should be global roots.

Venkatasubramanian et al. [Venkatasubramanian et al. 1992] assume a two-dimensional grid network topology, and the algorithm requires FIFO communication to flush communication channels. The FIFO communication assumption provides the ability to clear communication channels by a special *bulldoze* message, which is used for global snapshot. Any message sent in a cleared channel is marked as *new*. Inverse references are built during garbage collection for a distributed actor marking algorithm. The algorithm has five phases: 1) the pre-GC phase to determine a global snapshot, 2) the distributed scavenge phase to identify live actors in the snapshot, 3) the local-clear phase to notify each local collector to terminate the second phase, 4) the local-clear phase to reclaim garbage, and 5) the post GC broadcast phase to signal every computing node to terminate this cycle of global garbage collection.

Puaut proposed a snapshot-based algorithm in [Puaut 1994], which uses a server for global garbage collection. Each computing node maintains a timestamp vector to simulate a global clock. FIFO communication is assumed, but completely unordered communication is also possible by incorporation of timestamp-based message redelivery mechanism. Local garbage collectors send local reference graphs to a centralized server, as well as the timestamps of the last information received by the local garbage collectors. The server checks if the combined snapshot is consistent according to the timestamps. If this is not true, nothing is done. Otherwise the server performs global garbage collection, and then informs each local garbage collector the information of garbage. This approach is not scalable because the requirement to know every computing node to build the timestamp vector. Furthermore, it makes a message become larger while the number of computing nodes increases. The increase of the size of the system may also increase the chance of failure in global garbage collection.

Vardhan's algorithm [Vardhan 1998] transforms each local actor reference graph into a passive object reference graph, and uses Schelvis' algorithm [Schelvis 1989] for global garbage collection. It assumes: 1) First-In-First-Out (FIFO) communication, 2) temporarily suspending the message sender which is waiting for an acknowledgement from the message receiver, and 3) periodically performs stop-the-world garbage collection.

Wang and Varela [Wang and Varela 2006b] present a snapshot-based distributed actor garbage collection algorithm with proofs of correctness. The algorithm does not require First-In-First-Out or blocking communication, nor message logging. Actor migration is allowed while capturing global snapshots. Partial snapshots can be safely used to collect distributed garbage (cyclic or acyclic), therefore not requiring

comprehensive cooperation among all computing nodes. The algorithm assumes the live unblocked actor principle while capturing a consistent partial global view of the distributed system. Compared to other actor garbage collection algorithms, it is less intrusive, less restrictive, and more fault-tolerant to applications. The drawback of the algorithm is that it may not collect all kinds of active garbage (i.e. garbage that can mutate or migrate).

To conclude, existing actor garbage collection algorithms, except Wang-Varela snapshot-based algorithm, violate the asynchronous, non-FIFO assumption of actor communication, and do not support the concept of actor migration.

8. CONCLUSION AND FUTURE WORK

In this paper, we have redefined actor garbage to make the definition more operational. We introduced the concept of pseudo-roots, making actor GC easier to understand and to implement. The most important contribution of this paper is that *it formally describes the first reference listing algorithm for actor garbage collection, the pseudo-root approach, proving it correct*. Pseudo-root actor GC has been implemented in the open-source SALSA programming language [Worldwide Computing Laboratory 2007; Varela and Agha 2001]. Unlike existing actor GC algorithms, the proposed algorithm does not require FIFO communication or stop-the-world synchronization. Furthermore, it supports actor migration and it works concurrently with mutation operations. This feature reduces interruption of users' applications.

Our experimental results confirm that the pseudo-root approach is scalable and non-intrusive to applications. Together with the overhead of local actor garbage collection, the total overhead measured by different groups of operations is shown as follows:

(1) Actor creation: The overhead in a uniprocessor environment ranges from 23% to 58% while in a concurrent environment it is at most 22%.
(2) Actor migration: The overhead is negligible.
(3) Message passing (without any reference in messages): The overhead is negligible.
(4) Reference passing: The overhead in a uniprocessor environment is up to 104% while in a concurrent or distributed environment it becomes reasonable, ranging from 15% to 20%. These numbers show that the pseudo-root approach is practical in concurrent or distributed environments.

We also performed a scalability test using a real application, the MLE fitter. It is an iterative application which evaluates the state of the current iteration and then applying the evaluation results to the initial state of the next iteration. Parallelism is achieved by producing independent jobs at each iteration. Our experimental results show that the pseudo-root approach is scalable, and its total overhead (considering local GC) ranges from 1% to 15%, depending on the number of processors and the topology of the network.

The pseudo-root approach supports actor GC even if the computing system does not follow the live unblocked actor principle. Future research still needs to consider resource access restrictions, which are part of distributed resource management

policies. By applying resource access restrictions to actors, the live unblocked actor principle may not be true — not every actor has references to the root actors, and potentially live actors can be garbage. The migration privilege and the actor creation privilege are also important when resource access restrictions are considered. The reason is that they can change the semantics of actor garbage collection — an actor can migrate to a host which does not have resource access restrictions, or create a root actor to become live. Last but not least, testing the GC algorithms on larger-scale distributed environments is necessary to further evaluate their performance and scalability.

REFERENCES

Abdullahi, S. E. and Ringwood, A. 1998. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Computing Surveys 30,* 3, 330–373.

Agha, G. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press.

Armstrong, J., Virding, R., Wikström, C., and Williams, M. 1996. *Concurrent Programming in Erlang*, 2nd ed. Prentice Hall.

Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., and Quilici, R. 2006. *Grid Computing: Software Environments and Tools*. Springer-Verlag, Chapter Programming, Deploying, Composing, for the Grid.

Bevan, D. I. 1987. Distributed garbage collection using reference counting. In *PARLE'87*. Lecture Notes in Computer Science, vol. 258/259. Springer-Verlag, Eindhoven, The Netherlands, 176–187.

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

Blackburn, S. M., Hudson, R. L., Morrison, R., Moss, J. E. B., Munro, D. S., and Zigman, J. 2001. Starting with termination: a methodology for building distributed garbage collection algorithms. *Aust. Comput. Sci. Commun. 23,* 1, 20–28.

Briot, J.-P. 1989. Actalk: a testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'89)*. Cambridge University Press, 109–129.

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

Corporaal, H., Veldman, T., and van de Goor, A. 1990. An efficient reference weight-based garbage collection method for distributed systems. In *Proceedings of the PARBASE-90 Conference*. IEEE Press, Miami Beach, 463–465.

Cummings, J. P. and Weygand, D. P. 2003. An Object-Oriented Approach to Partial Wave Analysis. *ArXiv Physics e-prints*, 24 pp.

Dickman, P. 1996. Incremental, distributed orphan detection and actor garbage collection using graph partitioning and Euler cycles. In *WDAG'96*. Lecture Notes in Computer Science, vol. 1151. Springer-Verlag, Bologna.

Dijkstra, E. W. and Scholten, C. 1980. Termination detection for diffusing computations. *Information Processing Letters 11,* 1 (Aug.), 1–4.

El Maghraoui, K., Szymanski, B., and Varela, C. 2005. An architecture for reconfigurable iterative MPI applications in dynamic environments. In *Proc. of the Sixth International Conference on Parallel Processing and Applied Mathematics (PPAM'2005)*, R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, Eds. Number 3911 in LNCS. Poznan, Poland, 258–271.

Francez, N. 1980. Distributed termination. *ACM Trans. Program. Lang. Syst. 2,* 1, 42–55.

Hewitt, C. 1977. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence 8,* 3 (June), 323–364.

Hudson, R. L., Morrison, R., Moss, J. E. B., and Munro, D. S. 1997. Garbage collecting the world: One car at a time. *SIGPLAN Not. 32,* 10, 162–175.

Hughes, J. 1985. A distributed garbage collection algorithm. In *Record of the 1985 Conference on Functional Programming and Computer Architecture*. LNCS, vol. 201. Springer-Verlag, Nancy, France, 256–272.

Kafura, D., Mukherji, M., and Washabaugh, D. 1995. Concurrent and distributed garbage collection of active objects. *IEEE TPDS 6,* 4 (April).

Kafura, D., Washabaugh, D., and Nelson, J. 1990. Garbage collection of actors. In *OOPSLA'90 ACM Conference on Object-Oriented Systems, Languages and Applications*. ACM Press, 126–134.

Kim, W. 1997. THAL: An Actor System for Efficient and Scalable Concurrent Computing. Ph.D. thesis, University of Illinois at Urbana-Champaign.

Ladin, R. and Liskov, B. 1992. Garbage collection of a distributed heap. In *International Conference on Distributed Computing Systems*. Yokohama.

Lang, B., Queinnec, C., and Piquer, J. 1992. Garbage collecting the world. In *POPL'92 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 39–50.

Le Fessant, F. 2001. Detecting distributed cycles of garbage in large-scale systems. In *Principles of Distributed Computing (PODC)*. Rhodes Island.

Lermen, C. and Maurer, D. 1986. A protocol for distributed reference counting. In *ACM Symposium on Lisp and Functional Programming*. ACM SIGPLAN Notices. ACM Press, Cambridge, MA, 343–350.

Liskov, B. and Ladin, R. 1986. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth Annual ACM Symposium on the Principles on Distributed Computing*, J. Halpern, Ed. ACM Press, Calgary, 29–39.

Maheshwari, U. and Liskov, B. 1995. Collecting cyclic distributed garbage by controlled migration. In *PODC'95 Principles of Distributed Computing*.

Maheshwari, U. and Liskov, B. 1997. Collecting cyclic distributed garbage by back tracing. In *PODC'97 Principles of Distributed Computing*. ACM Press, Santa Barbara, CA, 239–248.

Matocha, J. and Camp, T. 1998. A taxonomy of distributed termination detection algorithms. *J. Syst. Softw. 43,* 3, 207–221.

Moreau, L. 2001. Tree rerooting in distributed garbage collection: Implementation and performance evaluation. *Higher-Order and Symbolic Computation 14,* 4, 357–386.

Moreau, L., Dickman, P., and Jones, R. 2005. Acm transactions on programming languages and systems (TOPLAS). *ACM Trans. on Program. Lang. and Syst. 27,* 6, 1344–1395.

Open Systems Lab. 1998. The Actor Foundry: A Java-based Actor Programming Environment. http://osl.cs.uiuc.edu/foundry/.

Piquer, J. M. 1991. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE'91*. Lecture Notes in Computer Science, vol. 505. Springer-Verlag, Eindhoven, The Netherlands.

Puaut, I. 1994. A distributed garbage collector for active objects. In *OOPSLA'94 ACM Conference on Object-Oriented Systems, Languages and Applications*. ACM Press, 113–128.

Rodrigues, H. and Jones, R. 1996. A cyclic distributed garbage collector for Network Objects. In *WDAG'96*. Lecture Notes in Computer Science, vol. 1151. Springer-Verlag, Bologna, 123–140.

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

Shapiro, M. 1991. A fault-tolerant, scalable, low-overhead distributed garbage collection protocol. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*. Pisa.

Shapiro, M., Dickman, P., and Plainfossé, D. 1992. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapports de Recherche 1799, INRIA. Nov.

Vardhan, A. 1998. Distributed garbage collection of active objects: A transformation and its applications to java programming. M.S. thesis, UIUC, Urbana Champaign, Illinois.

Vardhan, A. and Agha, G. 2002. Using passive object garbage collection algorithms. In *ISMM'02*. ACM SIGPLAN Notices. ACM Press, Berlin, 106–113.

Varela, C. A. and Agha, G. 2001. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 ACM Conference on Object-Oriented Systems, Languages and Applications 36,* 12 (Dec.), 20–34.

Veiga, L. and Ferreira, P. 2005. Asynchronous complete distributed garbage collection. In *IPDPS 2005*, O. Babaoglu and K. Marzullo, Eds. Denver, Colorado, USA.

Venkatasubramanian, N., Agha, G., and Talcott, C. 1992. Scalable distributed garbage collection for systems of active objects. In *IWMM'92*. Lecture Notes in Computer Science, vol. 637. Springer-Verlag.

Vestal, S. C. 1987. Garbage collection: An exercise in distributed, fault-tolerant programming. Ph.D. thesis, University of Washington, Seattle, WA.

Wang, W., Maghraoui, K. E., Cummings, J., Napolitano, J., Szymanski, B., and Varela, C. 2006. A middleware framework for maximum likelihood evaluation over dynamic grids. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*. Amsterdam, Netherlands, 8 pp.

Wang, W.-J. and Varela, C. A. 2006a. Distributed garbage collection for mobile actor systems: The pseudo root approach. In *Advances in Grid and Pervasive Computing, First International Conference, GPC 2006*. Lecture Notes in Computer Science, vol. 3947. Springer, 360–372.

Wang, W.-J. and Varela, C. A. 2006b. A non-blocking snapshot algorithm for distributed garbage collection of mobile active objects. Tech. Rep. 06-15, Dept. of Computer Science, R.P.I. Oct.

Watson, P. and Watson, I. 1987. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87*. Lecture Notes in Computer Science, vol. 258/259. Springer-Verlag, Eindhoven, The Netherlands, 432–443.

Worldwide Computing Laboratory. 2007. The SALSA Programming Language. http://wcl.cs.rpi.edu/salsa/.

Yonezawa, A. 1990. *ABCL An Object-Oriented Concurrent System*. MIT Press, Cambridge, Mass.